Jaime Evaristo

Aprendendo a Programar Programando na Linguagem C

Para Iniciantes

Terceira Edição
Revisada/Ampliada
Edição Digital
(cópias autorizadas)

Aprendendo
a
Programar
Programando
na
Linguagem C

Jaime Evaristo
Professor Adjunto
Instituto de Computação
Universidade Federal de Alagoas

Aos meus netos

Mateus, Vitor e Lucas

Sumário

1 Introdução à Programação

1.1 Organização básica de um computador

Um computador é constituído de quatro unidades básicas: *unidade de entrada*, *unidade de saída*, *unidade de processamento central* e *memória*. Como indica sua denominação, uma *unidade de entrada* é um dispositivo que permite que o usuário interaja com o computador, fornecendo-lhe dados e informações que serão processadas, sendo o *teclado* o seu exemplo mais trivial. Uma *unidade de saída*, por seu turno, serve para que sejam fornecidos ao usuário do computador os resultados do processamento realizado. O *monitor de vídeo* e uma *impressora* são exemplos de unidades de saída. A *unidade central de processamento* é responsável por todo o processamento requerido, sendo muito conhecida por ***cpu***, acrossemia de ***c**entral* ***p**rocessing* ***u**nit*. Já a *memória* armazena dados e informações que serão utilizados no processamento, armazenamento temporário, pois quando o computador é desligado tudo que está nela armazenado deixa de sê-lo (dizemos que toda a memória é "apagada").

1.2 Linguagem de máquina

Linguagens de comunicação

Evidentemente, há a necessidade de que as unidades que compõem um computador se comuniquem umas com as outra. Por exemplo, um dado fornecido pelo teclado deve ser armazenado na memória; para a *cpu* realizar uma operação aritmética, ela vai "buscar" valores que estão armazenados na memória, e assim por diante. Para que haja comunicação entre as unidades do computador é necessário que se estabeleça uma *linguagem*.

Os seres humanos se comunicam basicamente através de duas linguagens: a linguagem escrita e a fala. Uma comunicação através de uma linguagem escrita é constituída de *parágrafos*, os quais contêm *períodos*, que contêm *frases,* que são constituídas de *palavras*, sendo cada uma das palavras formadas por *letras* e esta sequência termina aí. Assim, uma *letra* é um ente indivisível da linguagem escrita e, em função disto, é chamada *símbolo básico* desta linguagem. Este exemplo foi apresentado para que se justifique a afirmação de que toda linguagem requer a existência de *símbolos básicos,* como - e para mais um exemplo - os *fonemas* para a linguagem falada.

A linguagem de comunicação entre as unidades

Como a comunicação entre as unidades do computador teria que ser obtida através de fenômenos físicos, os cientistas que conceberam os computadores atuais estabeleceram dois símbolos básicos para a linguagem. Esta quantidade de símbolos foi escolhida pelo fato de que através de fenômenos físicos é muito fácil obter dois estados distintos e não confundíveis, como passar corrente elétrica/não passar corrente elétrica, estar magnetizado/não estar magnetizado, etc., podendo cada um destes estados ser um dos símbolos. Assim a linguagem utilizada para comunicação interna num computador, chamada *linguagem de máquina*, possui apenas dois símbolos. Cada um destes símbolos é denominado *bit* (***bi**nary digi**t***) e eles são representados por 0 (zero) e 1 (um). Esta forma de representar os *bit's* justifica a sua denominação: *binary digit*, que significa dígito binário (além disto, *bit* em inglês significa fragmento). Portanto, as *palavras* da linguagem de máquina são sequências de bits, ou seja, sequências de dígitos zero e um.

O código ASCII

Para que haja a possibilidade da comunicação do homem com o computador, é necessário que as palavras da linguagem escrita sejam traduzidas para a linguagem de máquina e vice-versa. Para que isto seja possível, é necessário que se estabeleça qual a sequência de bit's que corresponde a cada caractere usado na linguagem escrita. Ou seja, é necessário que se estabeleça uma codificação em sequência de bit's para cada um dos caracteres. Uma codificação muito utilizada é o *código ASCII* (*American Standard Code for Information Interchange* ou *Código Padrão Americano para Intercâmbio de Informações*), estabelecido pelo *ANSI* (*American National Standards Institute*). Nesta codificação, cada caractere é representado por uma sequência de oito bits (normalmente, um conjunto de oito bit's é chamado *byte*). Só para exemplificar (será visto ao longo do livro que, em geral, não há necessidade de que se conheça os códigos dos caracteres), apresentamos a tabela abaixo com os códigos ASCII de alguns caracteres.

Tabela 1 Códigos ASCII de alguns caracteres

Caractere	Código ASCII
Espaço em branco	00100000
!	00100001
"	00100010
. . .	. . .
0	00110000
1	00110001
. . .	. . .
A	01000001
B	01000010
. . .	. . .
Z	01011010
. . .	. . .
a	01100001
. . .	. ..

Observe a necessidade de se haver codificado o *espaço em branco* (este "caractere" é utilizado para separar nossas palavras) e de se haver codificado diferentemente as letras maiusculas e minúsculas, para que se possa considerá-las como coisas distintas.

Levando em conta que cada sequência de zeros e uns pode ser vista como a representação de um número inteiro no *sistema binário de numeração* [Evaristo, J 2002], podemos, até para facilitar a sua manipulação, associar a cada código ASCII o inteiro correspondente, obtendo assim o que se costuma chamar de *código ASCII decimal*. Por exemplo, como 1000001 é a representação do número (decimal) 65 no sistema binário de numeração, dizemos que o *código ASCII decimal* de A é 65.

1.3 Programas de computadores

Para que um computador tenha alguma utilidade, ele deve executar um *programa* que tenha uma finalidade específica. *Games* são programas que têm como objetivo propiciar entretenimento aos seus usuários. *Processadores de texto* são programas que permitem que textos sejam digitados, impressos e armazenados para futuras modificações ou impressões. *Planilhas eletrônicas* são programas que oferecem recursos para manipulação de tabelas de valores numéricos. *Navegadores* permitem acessos a páginas da *internet*, a rede mundial de computadores. Estes programas destinam-se a *usuários finais*, aquelas pessoas que vão utilizar o computador com um determinado objetivo específico, usando para tal um programa que ela aprendeu a usar, não tendo nenhuma preocupação relativa ao funcionamento interno do sistema computador/programa. Por exemplo, um usuário de um processador de texto deve aprender o que fazer para que o processador destaque em **negrito** alguma parte do texto ou localize uma palavra, não havendo necessidade de saber como o programa realiza estas ações.

Na verdade, para que um processador de texto propicie ao usuário a possibilidade de que textos sejam digitados, corrigidos, gravados, inseridos em outros textos e de que palavras sejam localizadas dentro de um

texto, é necessária a execução de muitas instruções com objetivos bem mais específicos e restritos. Um *programa de computador* é, na realidade, um conjunto de instruções que podem ser executadas pelo computador, de tal forma que a execução de subconjuntos destas instruções permitem a realização de ações mais genéricas.

É muito grande o número de instruções dos programas citados acima, chegando à casa dos milhares. Rigorosamente falando, um programa dos acima citados são conjunto de programas menores, cada um deles com objetivos mais restritos, e que podem ser executados de forma integrada. É comum se utilizar a palavra inglesa *software* para designar um conjunto de programas com objetivos mais restritos que, sendo executados de forma integrada, propiciam a execução de ações bem mais genéricas.

A parte da Ciência da Computação que trata do desenvolvimento de *softwares* é denominada *Engenharia de Software*. Naturalmente, o estudo da *Engenharia de Software* deve ser precedido da aprendizagem do desenvolvimento de programas "menores", ação que comumente é denominada de *Programação de Computadores*.

1.4 Lógica de programação

Sendo um conjunto de instruções cujas execuções redundam na realização da tarefa para a qual foi desenvolvido, o desenvolvimento de um programa requer a utilização de um raciocínio ímpar em relação aos raciocínios utilizados na solução de problemas de outros campos do saber. Por exemplo (e de forma simplificada) ao se tentar resolver um problema de Mecânica Newtoniana deve-se procurar capturar da especificação da questão as grandezas físicas envolvidas e aplicar as fórmulas que relacionam estas grandezas.

Para se desenvolver um programa que resolva um determinado problema é necessário que encontremos uma sequência de instruções que cujas execuções resultem na solução da questão. É comum se utilizar a termo *algoritmo* para indicar uma sequência de instruções que resolvem um dado problema, ficando, neste caso, o termo *programa* para indicar um algoritmo que pode ser executado num computador. A *Lógica de Programação* pode ser entendida como o conjunto de raciocínios utilizados para o desenvolvimento de algoritmos (e, portanto, de programas).

Por exemplo, imagine a seguinte questão: um senhor, infelizmente bastante gordo, está numa das margens de um rio com uma raposa, uma dúzia de galinhas e um saco de milho. O senhor pretende atravessar o rio com suas cargas, num barco a remo que só comporta o senhor e uma das cargas. Evidentemente, o senhor não pode deixar em uma das margens, sozinhos, a raposa e a galinha, nem a galinha e o milho. A questão é escrever um algoritmo que oriente o senhor a realizar o seu intento. Naturalmente, na primeira viagem, ele não pode levar a raposa (neste caso, as galinhas comeriam o milho), nem o milho (caso em que a raposa devoraria as galinhas). Logo, na primeira viagem ele deve levar as galinhas. Como ele estará presente na chegada, na segunda viagem ele pode levar a raposa ou o milho. Mas, e a volta para apanhar terceira carga? A solução é ele voltar com as galinhas e, aí, atravessar o milho, já que não há problema em que a raposa e o milho fiquem juntos. Escrevendo as instruções na sequência em que elas devem ser executadas, teremos o seguinte algoritmo.

1. Atravesse as galinhas.
2. Retorne sozinho.
3. Atravesse a raposa.
4. Retorne com as galinhas.
5. Atravesse o milho.
6. Retorne sozinho.
7. Atravesse as galinhas.

1.5 Resolução de problemas

Uma pergunta que o leitor pode estar se fazendo é: como vou "descobrir" que a primeira instrução deve ser a travessia das galinhas?

Algumas tarefas para as quais se pretende escrever um algoritmo podem ser vistas como um problema

a ser resolvido. O exemplo anterior é um exemplo claro de uma tarefa com esta característica. Existem algumas técnicas que podem ser utilizadas para a resolução de problemas. No exemplo anterior, para se definir qual seria a primeira instrução, como existem apenas três possibilidades, verifica-se o que aconteceria ao se escolher determinada instrução. Foi o que, de passagem, foi feito acima: se o homem atravessasse primeiro o milho, a raposa devoraria as galinhas; se o homem atravessasse primeiro a raposa, as galinhas comeriam o milho. Neste caso, podemos dizer que foi utilizada a técnica da exaustão: como o número de alternativas era pequeno, analisamos todas elas, uma a uma.

Esta técnica pode ser utilizada também na solução do seguinte problema: dispõe-se de três esferas idênticas na forma, sendo duas delas de mesmo peso e a terceira de peso maior. A questão é descobrir qual a esfera de peso diferente, realizando-se apenas uma pesagem numa balança de dois pratos. Para isto chamemos de A e B as esferas de mesmo peso e de C a de maior peso. Se optarmos por colocar duas esferas num dos pratos e a outra esfera no outro, temos as seguintes possibilidades:

a) (A+B, C).
b) (A+C, B).
c) (B+C, A).

No primeiro caso, pode acontecer qualquer coisa: a balança pode ficar equilibrada, se Peso(C) = Peso(A+B); ficar inclinada para o lado esquerdo, se Peso(C) > Peso(A+B) ou ficar inclinada para o lado direito se Peso(C) < Peso(A+B). Observe que nada pode distinguir a esfera C. Nos dois últimos casos, a balança se inclinará para a esquerda, mas, outra vez, nada distingue a esfera C. Por exaustão, resta então escolhermos duas esferas e colocarmos cada uma delas num dos pratos da balança. Agora os casos possíveis são:

a) (A, B).
b) (A, C).
c) (B, C).

No primeiro caso, a balança ficará equilibrada, o que indica que a mais pesada é aquela não escolhida; nos outros dois casos, a balança se inclinará para a direita, indicando que a esfera mais pesada é aquela que ocupa o prato respectivo. Temos então o seguinte algoritmo:

1. Escolha duas esferas.
2. Coloque cada uma das esferas escolhidas num dos pratos da balança.
3. Se a balança ficar equilibrada, forneça como resposta a esfera não escolhida; caso contrário, forneça como resposta a esfera do prato que está num nível mais baixo.

Uma outra técnica de resolução de problemas consiste em se tentar resolver casos particulares da questão ou resolver a questão para dados menores do que os dados que foram fixados. Para exemplificar, consideremos a seguinte questão: como obter exatamente 4 litros de água dispondo de dois recipientes com capacidades de 3 litros e 5 litros[1]? Como 4 = 3 + 1 ou 4 = 5 – 1 conseguiremos resolver a questão se conseguirmos obter 1 litro. Mas isto é fácil, pois 1 = 3 + 3 – 5! Temos então o seguinte algoritmo:

1. Encha o recipiente de 3 litros.
2. Transfira o conteúdo do recipiente de 3 litros para o recipiente de 5 litros.
3. Encha o recipiente de 3 litros.
4. Com o conteúdo do recipiente de 3 litros, complete o recipiente de 5 litros.
5. Esvazie o recipiente de 5 litros.
6. Transfira o conteúdo do recipiente de três litros para o recipiente de 5 litros.
7. Encha o recipiente de 3 litros.
8. Transfira o conteúdo do recipiente de 3 litros para o recipiente de 5 litros.

Para compreender o algoritmo, sejam A e B os recipientes de 3 litros e de 5 litros, respectivamente, e indiquemos por (X, n) o fato de o recipiente *X* conter *n* litros de água. No início temos (A, 0) e (B, 0) e, após a execução de cada instrução, teremos:

1. (A, 3), (B, 0).
2. (A, 0), (B, 3).
3. (A, 3), (B, 3).
4. (A, 1), (B, 5).

[1] A solução desta questão foi necessária num filme da série Duro de Matar para o protagonista desativar uma bomba.

5. (A, 1), (B, 0).
6. (A, 0), (B, 1).
7. (A, 3), (B, 1).
8. (A, 0), (B, 4).

Outras questões que podem ser levantadas são: há outras soluções? Existe alguma solução que realize a mesma tarefa com uma quantidade menor de instruções? Para responder a estas questões talvez seja interessante lembrar que 4 = 5 – 1. Significa que, se conseguirmos tirar 1 litro do recipiente de 5 litros quando ele estiver cheio, resolveremos a questão. Para conseguir isto, basta que o recipiente de 3 litros contenha 2 litros. E para se obter 2 litros? Aí basta ver que 2 = 5 – 3. Podemos então resolver a questão com o seguinte algoritmo, constituído de apenas seis instruções:

1. Encha o recipiente de 5 litros.
2. Com o conteúdo do recipiente de 5 litros, encha o de 3 litros.
3. Esvazie o recipiente de 3 litros.
4. Transfira o conteúdo do recipiente de 5 litros para o recipiente de 3 litros.
5. Encha o recipiente de 5 litros.
6. Com o conteúdo do recipiente de 5 litros, complete o recipiente de 3 litros.

Após a execução de cada uma das instruções teremos:

1. (A, 0), (B, 5).
2. (A, 3), (B, 2).
3. (A, 0), (B, 2).
4. (A, 2), (B, 0).
5. (A, 2), (B, 5).
6. (A, 3), (B, 4).

Uma outra técnica bastante utilizada é se tentar raciocinar a partir de uma solução conhecida de uma outra questão. Para compreender isto considere as duas seguintes questões: imagine uma relação de *n* números, os quais podem ser referenciados por a_i com i = 1, 2, ..., n e queiramos somá-los com a restrição de que só sabemos efetuar somas de duas parcelas. Para resolver esta questão, podemos pensar em casos particulares: se n = 2, basta somar os dois números; se n = 3, basta somar os dois primeiros e somar esta soma com o terceiro. Naturalmente este raciocínio pode ser reproduzido para n > 3. A questão é que a soma dos dois primeiros deve estar "guardada" para que se possa somá-la com o terceiro, obtendo-se a soma dos três primeiros; esta soma deve ser "guardada" para que seja somada com o quarto e assim sucessivamente. Para isto podemos estabelecer uma *referência* à soma "atual", a qual será alterada quando a soma com o elemento seguinte for efetuada. Até para somar os dois primeiros, pode-se pensar em somar "a soma do primeiro" com o segundo.

Temos então o seguinte algoritmo:

1. Faça i = 1.
2. Faça Soma = a_1.
3. Repita n – 1 vezes as instruções 3.1 e 3.2.
   3.1. Substitua i por i + 1.
   3.2. Substitua Soma por Soma + a_i.

Por exemplo: se n = 5 e a_1 = 8, a_2 = 4, a_3 = 9, a_4 = 13 e a_5 = 7, a execução do algoritmo resultaria nas seguintes ações:

1. i = 1.
2. Soma = 8.
3.1.1. i = 2.
3.2.1. Soma = 8 + 4 = 12
3.1.2. i = 3.
3.2.2. Soma = 12 + 9 = 21.
3.1.3. i = 4.
3.2.3. Soma = 21 + 13 = 34.
3.1.4. i = 5.
3.2.4. Soma = 34 + 7 = 41.

Naturalmente, na execução acima estamos indicando por 3.1.x e 3.2.x a execução de ordem x das

instruções 3.1 e 3.2.

Como veremos ao longo do livro, este algoritmo é bastante utilizado em programação, sendo mais comum até o primeiro termo da relação ser "somado" dentro da repetição. Neste caso, para que o primeiro seja somado, é necessário que *Soma* seja *inicializado* com 0 (zero), ficando assim o algoritmo:

1. Faça i = 0.
2. Faça Soma = 0.
3. Repita n vezes as instruções 3.1 e 3.2.
   3.1. Substitua i por i + 1.
   3.2. Substitua Soma por Soma + a_i.

Conhecendo este algoritmo, é fácil então resolver a questão de se calcular o produto de *n* números nas mesmas condições, e aí vemos como utilizar uma solução conhecida para resolver um problema. Deve-se inicializar uma referência *Produto* com 1 e, numa repetição, multiplicar os números como foi feito no caso da soma:

1. Faça i = 0.
2. Faça Produto = 1.
3. Repita n vezes as instruções 3.1 e 3.2.
   3.1. Substitua i por i + 1.
   3.2. Substitua Produto por Produto x a_i.

1.6 Processador de um algoritmo

Obviamente, um algoritmo deve ser executado por algum agente. Este agente pode ser uma pessoa munida de certos equipamentos e utensílios ou por máquinas projetadas para executar automaticamente algumas instruções básicas. O algoritmo para a travessia do senhor gordo com as galinhas, sua raposa e seu saco de milho seria executado pelo tal senhor, que estava para tal munido do barco e de remos. O algoritmo para obtenção de quatro litros de água a partir de recipientes de conteúdos cinco litros e três litros poderia ser executado por uma pessoa que dispusesse dos dois recipientes e de água em abundância. Neste último caso, quem sabe, a pessoa poderia ser substituída por um robô.

O agente que executa um algoritmo é chamado *processador* e é evidente que para que o algoritmo seja executado é necessário que o processador seja capaz de executar cada uma das suas instruções. Se o senhor gordo não souber remar ele não será capaz de atravessar o rio. Uma pessoa que não seja capaz de esvaziar um recipiente que pese cinco quilos não será capaz de executar o algoritmo dos quatro litros de água.

Alguns autores de livros com objetivos idênticos a este - facilitar a aprendizagem da programação de computadores - iniciam seus textos discorrendo exclusivamente sobre *resolução de problemas*, encarando o processador como uma "caixa preta" que recebe as instruções formuladas pelo algoritmo e fornece a solução do problema, não levando em conta o processador quando da formulação do tal algoritmo. Entendemos que esta não é a melhor abordagem, visto que o conhecimento do que o processador pode executar pode ser definidor na elaboração do algoritmo. Por exemplo: imagine que queiramos elaborar um algoritmo para extrair o algarismo da casa das unidades de um inteiro dado (apresentaremos posteriormente uma questão bastante prática cuja solução depende deste algoritmo). Evidentemente, o algoritmo para resolver esta “grande” questão depende do processador que vai executá-lo. Se o processador for um ser humano que saiba o que é número inteiro, algarismo e casa das unidades, o algoritmo teria uma única instrução:

1. Forneça o algarismo das unidades do inteiro dado.

Porém, se o processador for um ser humano que saiba o que é número inteiro e algarismo, mas não saiba o que é casa das unidades, o algoritmo não poderia ser mais esse. Neste caso, para resolver a questão, o processador deveria conhecer mais alguma coisa, como, por exemplo, ter a noção de "mais à direita", ficando o algoritmo agora como:

1. Forneça o algarismo "mais à direita" do número dado.

E se o processador é uma máquina e não sabe o que é algarismo, casa das unidades, "mais à direita", etc.? Nesta hipótese, quem está elaborando o algoritmo deveria conhecer que instruções o processador é capaz de executar para poder escrever o seu algoritmo. Por exemplo, se a máquina é capaz de determinar o resto de uma divisão inteira, o algoritmo poderia ser:

1. Chame de n o inteiro dado;
2. Calcule o resto da divisão de n por 10;
3. Forneça este resto como o algarismo pedido.

Algumas das questões anteriores são importantes para se desenvolver o raciocínio, mas não é este tipo de questão que se pretende discutir ao longo deste livro. Estamos interessados em algoritmos para:

1. Resolver problemas matemáticos, como algoritmos para determinar a média aritmética de vários números dados, determinar as raízes de uma equação do segundo grau, encontrar o máximo divisor comum de dois números dados, totalizar as colunas de uma tabela, etc.
2. Resolver questões genéricas, como algoritmos para colocar em ordem alfabética uma relação de nomes de pessoas, atualizar o saldo de uma conta bancária na qual se fez um depósito, corrigir provas de um teste de múltipla escolha, cadastrar um novo usuário de uma locadora, etc..

Na linguagem coloquial, o algoritmo para o cálculo da média pode ser escrito de forma muito simples:

1. Determine a quantidade de números;
2. Some os números dados;
3. Divida esta soma pela quantidade de números.

Qualquer pessoa que saiba contar, somar e dividir números é capaz de executar este algoritmo dispondo apenas de lápis e papel. A questão que se põe é: e se a relação contiver 13.426 números? A tal pessoa é capaz de executar, porém, quanto tempo levará para fazê-lo?

Um outro aspecto a ser observado é que nem sempre a linguagem coloquial é eficiente para se escreverem as instruções. Nessa linguagem o algoritmo para determinação das raízes de uma equação do segundo grau teria uma instrução difícil de escrever e difícil de compreender como:

n. Subtraia do quadrado do segundo coeficiente o produto do número quatro pelo produto dos dois outros coeficientes.

Isto pode ser parcialmente resolvido utilizando-se uma linguagem próxima da linguagem matemática que já foi utilizada em exemplos da seção anterior. No caso da equação do segundo grau teríamos o seguinte algoritmo, que nos é ensinado nas últimas séries do ensino fundamental:

1. Chame de a, b e c os coeficientes da equação.
2. Calcule $d = b^2 - 4ac$.
3. Se $d < 0$ forneça como resposta a mensagem: A equação não possui raízes reais.
4. Se $d \geq 0$
   4.1 Calcule $x_1 = (-b + raiz(d))/2a$ e $x_2 = (-b - raiz(d))/2a$.
   4.2 Forneça x_1 e x_2 como raízes da equação.

De maneira mais ou menos evidente, *raiz(d)* está representando a raiz quadrada de *d* e a execução deste algoritmo requer que o processador seja capaz de determinar valores de expressões aritméticas, calcular raízes quadradas, efetuar comparações e que conheça a linguagem matemática.

Algoritmos para problemas genéricos são mais complicados e as linguagens utilizadas anteriormente não são adequadas (para o caso da ordenação de uma relação de nomes, foram desenvolvidos vários algoritmos e teremos oportunidade de discutir alguns deles ao longo deste livro).

1.7 Exemplos de algoritmos matemáticos

Para uma primeira discussão em termos de aprendizagem de desenvolvimento de algoritmos e utilizando a linguagem usada no exemplo da equação do segundo grau, apresentamos a seguir alguns exemplos de algoritmos que objetivam a solução de questões da matemática. Para eles supomos que o processador seja capaz de efetuar somas, subtrações e divisões decimais, de realizar comparações, de repetir a execução de um conjunto de instruções um número determinado de vezes ou enquanto uma condição seja atendida.

1. No exemplo do algoritmo para obtenção do algarismo da casa das unidades de um inteiro dado supomos que o processador seria capaz de calcular o resto de uma divisão inteira. Observando que não está suposto que o nosso processador seja capaz de determinar restos de divisões inteiras, vamos discutir um

algoritmo para a determinação do quociente e do resto da divisão de dois inteiros positivos dados. Por exemplo: se o dividendo for 30 e o divisor for 7, o algoritmo deve fornecer os valores 4 para o quociente e 2 para o resto. Fomos ensinados que, para determinar o quociente, deveríamos, por tentativa, encontrar o número que multiplicado pelo divisor resultasse no maior número menor que o dividendo. No exemplo numérico citado, poderíamos tentar o 5 e teríamos 5x7 = 35 que é maior que 30; tentaríamos o 3 obtendo 3x7 = 21 que talvez seja pequeno demais em relação ao 30; aí tentaríamos o 4 obtendo 4x7 = 28, encontrando então o quociente 4. Um algoritmo para solucionar esta questão poderia ser:

1. Chame de D1 e D2 o dividendo e o divisor dados.
2. Faça I = 1.
3. repita 3.1 até IxD2 > D1.
   3.1. Substitua I por I + 1.
4. Calcule Q = I – 1.
5. Calcule R = D1 - QxD2.
6. Forneça R para o resto e Q para o quociente pedidos.

No exemplo numérico proposto, teríamos a seguinte tabela com os valores obtidos durante a execução do algoritmo:

D1	D2	I	QxI	Q	R
30	7				
		1	7		
		2	14		
		3	21		
		4	28		
		5	35		
				4	2

2. O algoritmo abaixo determina o menor divisor maior que 1 de um inteiro dado. A ideia é verificar se d = 2 é divisor e, não sendo, verificar se 3 ou 4 ou 5, etc, é divisor. A procura por um divisor vai até que um divisor seja encontrado. Naturalmente, utilizando o algoritmo anterior, o nosso processador agora sabe determinar o resto da divisão inteira de um inteiro x por outro inteiro y não nulo. Isto será indicado por Resto(x, y). Para encontrar um divisor de n basta encontrar um inteiro d tal que Resto(n, d) = 0.

1. Chame de N o inteiro dado.
2. Faça D = 2.
3. Repita 3.1 enquanto Resto(N, D) ≠ 0
   3.1 Substitua D por D + 1
4. Forneça D para o divisor procurado.

3. Como se depreende facilmente da sua denominação, o *máximo divisor comum* (*mdc*) de dois números dados é o maior número que os divide. Antes o *mdc* só era utilizado para simplificações de frações ordinárias; atualmente ele é utilizado na determinação de *chaves públicas* para sistemas de criptografia RSA [Evaristo, J, 2002]. Por exemplo, mdc(64, 56) = 8. De maneira óbvia, o algoritmo abaixo determina o *mdc* de dois números dados:

1. Chame de x e de y os números.
2. Determine D(x), o conjunto dos divisores de x.
3. Determine D(y), o conjunto dos divisores de y.
4. Determine I, a interseção de D(x) e D(y).
5. Determine M, o maior elemento do conjunto I.
6. Forneça M como o mdc procurado.

O cálculo de mdc(120, 84) com este algoritmo seria:

1. x = 120, y = 84.
2. D(120) = {1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, 60, 120}.
3. D(84) = {1, 2, 3, 4, 6, 7, 12, 14, 21, 28, 42, 84}.
4. I = {1, 2, 3, 4, 6, 12}.
5. M = 12.

Observe que o algoritmo anterior determina o menor divisor de um inteiro não determinando todos os

divisores, como necessário neste exemplo. Observe também que estamos supondo que o nosso processador é capaz de determinar a interseção de dois conjuntos.

A matemática fornece uma outra forma de se calcular o *mdc* de dois inteiros: determina-se a *decomposição em fatores primos* dos dois inteiros e o *mdc* é o produto dos fatores primos comuns as duas decomposições com as menores multiplicidades.

Para o exemplo dado acima teríamos:

120	2		84	2
60	2		42	2
30	2		21	3
15	3		7	7
5	5		1	
1	1			

o que nos dá $120 = 2^3x3x5$ e $84 = 2^2x3x7$ e, portanto, mdc(120, 84) = $2^2x3 = 12$.

Vale observar que escrever este algoritmo na linguagem informal que estamos utilizando é bastante complicado.

Na há dúvida que o primeiro algoritmo para o cálculo do *mdc* apresentado é de compreensão bastante simples. Porém, comentaremos posteriormente que ele é computacionalmente bastante ineficiente no sentido de que sua execução pode, dependendo dos valores de *x* e *y*, demandar um tempo acima do razoável.

Por incrível que possa parecer, o algoritmo mais eficiente para o cálculo do máximo divisor comum de dois números foi desenvolvido pelo matemático grego Euclides duzentos anos Antes de Cristo. O *algoritmo de Euclides* nos é apresentado nas séries intermediárias do ensino fundamental através de um esquema como o diagrama do exemplo abaixo, cujo objetivo é determinar (de novo!) o máximo divisor comum de 120 e 84.

	1	2	3
120	84	36	12

O esquema funciona da seguinte forma: divide-se 120 por 84 obtendo-se resto 36; a partir daí, repetem-se divisões até que o resto seja zero, sendo o dividendo da divisão atual o divisor da divisão anterior e o divisor da divisão atual o resto da divisão anterior. O último divisor é o máximo divisor procurado. Como se pode ver, estas instruções escritas desta forma não são nada compreensíveis, o que faz com elas sejam transmitidas oralmente nas salas do ensino fundamental. No capítulo 4 (quatro), teremos a oportunidade de discutir este algoritmo com detalhes e veremos que ele é um algoritmo bastante interessante no desenvolvimento da lógica de programação.

4. Discutiremos agora o algoritmo para o cálculo da média de uma relação contendo um número grande (digamos, 10 000) de números dados. No caso da equação do segundo grau, eram três os dados de entrada e, portanto, os chamamos de *a*, *b*, e *c.* Mas agora são 10 000 os dados de entrada! Uma solução possível é receber os números um a um, somando-os antes de receber o seguinte, conforme vimos na seção 1.5.

1. Chame de A o primeiro número dado.
2. Faça S = A.
3. Repita 9 999 vezes as instruções 3.1 e 3.2.
   3.1 Chame de A o próximo número dado.
   3.2 Substitua o valor de S por S + A.
4. Calcule M = S/10 000.
5. Forneça M para o valor da média.

Por exemplo, se a relação de números fosse {5, 3, 8, 11, ...} até a quarta execução de 3.1 e 3.2 teríamos a seguinte tabela:

A	S	M
5	5	
3	8	
8	16	
11	27	

Está fácil perceber que após 9.999ª execução das instruções 3.1 e 3.2 a variável S conterá a soma de todos os números da relação, o que justifica a instrução 4.

5. Um outro exemplo que justifica plenamente a necessidade do conhecimento do que o processador é capaz de executar é a determinação do maior número de uma relação de números. Se o processador for um

aluno do ensino médio e a relação contiver poucos números, uma simples olhada na relação permitirá se identificar o maior número. Mas, e se o processador for um aluno das classes iniciais do ensino fundamental? E se a relação contiver 10 000 números? E se os números estiverem escritos em forma de fração ordinária?

Uma solução possível é supor que o maior número é o primeiro da relação e comparar este suposto maior com os demais números, alterando-o quando for encontrado um número na relação maior do que aquele que até aquele momento era o maior.

1. Chame de A o primeiro número dado.
2. Faça M = A.
3. Repita 9 999 vezes as instruções 3.1 e 3.2.
   3.1 Chame de A o próximo número dado.
   3.2 Se A > M substitua o valor de M por A.
4. Forneça M para o valor do maior número.

Para exemplificar, suponha que a entrada fosse o conjunto {5, 3, 8, 11, 10...}. Até a quinta execução das instruções 3.1 e 3.2 teríamos a seguinte tabela:

A	M
5	5
3	
8	8
11	11
10	

1.8 Linguagens de alto nível

Computadores digitais foram concebidos para executarem instruções escritas em linguagem de máquina. Isto significa que um computador é capaz de executar um algoritmo escrito nesta linguagem. Um algoritmo escrito em linguagem de máquina é normalmente chamado de *programa objeto*. Nos primórdios da computação, os algoritmos que se pretendiam que fossem executados por um computador eram escritos em linguagem de máquina, o que tornava a tarefa de desenvolvimento de algoritmos muito trabalhosa, devido ao fato de que era necessário que se conhecesse qual sequência de bits correspondia à instrução pretendida. Naturalmente, esta dificuldade acontecia pelo fato de que o ser humano não está habituado a uma linguagem com apenas dois símbolos básicos. Um grande avanço ocorreu na computação quando se conseguiu desenvolver programas que traduzissem instruções escritas originariamente numa linguagem dos seres humanos para a linguagem de máquina. O surgimento de programas para esta finalidade permitiu o desenvolvimento de algoritmos em linguagens que utilizam caracteres, palavras e expressões de um idioma, ou seja, uma linguagem cujos símbolos básicos e cujas palavras estão no cotidiano do ser humano. Uma linguagem com esta característica é chamada *linguagem de alto nível*, onde *alto nível* aí não se refere à qualidade e sim ao fato de que ela está mais próxima da linguagem do ser humano do que da linguagem da máquina (quando alguma coisa está mais próxima da máquina do que do ser humano dizemos que ela é *de baixo nível*). Como exemplos de linguagens de alto nível temos *Pascal*, *C*, *Delphi*, *Visual Basic, Java* e C^{++}. Um algoritmo escrito numa linguagem de alto nível é chamado *programa fonte* ou simplesmente *programa*

Como foi dito acima, um *programa fonte* deve ser traduzido para a linguagem de máquina. Há dois tipos de programas que fazem isto: os *interpretadores* que traduzem os comandos para a linguagem de máquina um a um e os *compiladores* que traduzem todo o programa para a linguagem de máquina. Um compilador ao receber como entrada um programa fonte fornece como saída um programa escrito em linguagem de máquina, chamado *programa objeto*. A *compilação* do programa, portanto, gera um programa que pode então ser executado. É comum nos referirmos à execução do programa fonte quando se está executando o programa objeto.

Já um interpretador traduz para a linguagem de máquina os comandos do programa um a um, executando-os em seguida. Assim a *interpretação* de um programa não gera um programa objeto.

1.9 Sintaxe e semântica de uma instrução

O que é sintaxe

Dissemos que um programa escrito em linguagem de alto nível é traduzido para a linguagem de máquina por um compilador ou cada instrução é traduzida por um interpretador. É natural se admitir que, para que o compilador consiga traduzir uma instrução escrita com caracteres de algum idioma para instruções escritas como sequências de zeros e uns, é necessário que cada instrução seja escrita de acordo com regras preestabelecidas. Estas regras são chamadas *sintaxe* da instrução e quando não são obedecidas dizemos que existe *erro de sintaxe*.

Se o programa fonte contém algum erro de sintaxe, o compilador não o traduz para a linguagem de máquina (isto é, o compilador não *compila* o programa) e indica qual o tipo de erro cometido e a instrução onde este erro aconteceu. Se o programa fonte for interpretado, ele é executado até a instrução que contém o erro, quando então é interrompida a sua execução e o tal erro é indicado.

O que é semântica

Naturalmente, cada instrução tem uma finalidade específica. Ou seja, a execução de um instrução resulta na realização de alguma ação, digamos *parcial*, e é a sequência das ações parciais que redunda na realização da tarefa para a qual o programa foi escrito. A ação resultante da execução de uma instrução é chamada *semântica* da instrução. Infelizmente, um programa pode não conter erros de sintaxe (e, portanto, pode ser executado), mas a sua execução não fornecer como saída o resultado esperado para alguma entrada. Neste caso, dizemos que o programa contém *erros de lógica* que, ao contrário dos erros de sintaxe que são detectados pelo compilador ou pelo interpretador, são, às vezes, de difícil detecção.

No nosso entendimento, para aprender a programar numa determinada linguagem é necessário que se aprenda as instruções daquela linguagem (para que se conheça o que o processador é capaz de fazer), a sintaxe de cada um destes instruções e as suas semânticas. Aliado a isto, deve-se ter um bom desenvolvimento de *lógica programação* para que se escolha as instruções necessárias e a sequência segundo a qual estas instruções devem ser escritas, para que o programa, ao ser executado, execute a tarefa pretendida. Felizmente ou infelizmente, para cada tarefa que se pretende não existe apenas uma sequência de instruções que a realize. Ou seja, dado um problema não existe apenas um programa que o resolva. Devemos procurar o *melhor programa*, entendendo-se como *melhor programa* um programa que tenha boa *legibilidade*, cuja execução demande o menor tempo possível e que necessite, para sua execução, a utilização mínima da memória.

Existe um conjunto de instruções que é comum a todas as linguagens de alto nível e cujas semânticas permitem executar a maioria das tarefas. A aprendizagem das semânticas destas instruções e das suas sintaxes em alguma linguagem de programação (aliado ao desenvolvimento da *lógica de programação*) permite que se aprenda com facilidade outra linguagem do mesmo paradigma.

1.10 Sistemas de computação

Como foi dito anteriormente, a *cpu* de um computador é capaz de executar instruções (escritas em linguagem de máquina, permitam a repetição). Ou seja, um computador é capaz de executar programas e só para isto é que ele serve. Se um computador não estiver executando um programa ele para nada está servindo. Como foram concebidos os computadores atuais, um programa para ser executado deve estar armazenado na sua *memória*. O armazenamento dos programas (e todo o gerenciamento das interações entre as diversas unidades do computador) é feito por um programa chamado *sistema operacional*. Um dos primeiros sistemas operacionais para gerenciamento de microcomputadores foi o *DOS (Disk Operating System)*. Quando um computador é ligado, de imediato o sistema operacional é armazenado na memória e só a partir daí o computador está apto a executar outros programas. Estes programas podem ser um *game*, que

transforma o "computador" num poderoso veículo de entretenimento; podem ser um *processador de texto*, que transforma o "computador" num poderoso veículo de edição de textos; podem ser uma *planilha eletrônica*, que transforma o "computador" num poderoso veículo para manipulação de tabelas numéricas, podem ser programas para gerenciar, por exemplo, o dia a dia comercial de uma farmácia e podem ser ambientes que permitam o desenvolvimento de *games* ou de programas para gerenciar o dia a dia comercial de uma farmácia. Talvez com exceção de um *game*, os programas citados acima são, na verdade, conjuntos de programas que podem ser executados de forma integrada. Um conjunto de programas que podem ser executados de forma integrada é chamado *software*. Por seu turno, as unidades do computador, associadas a outros equipamentos chamados *periféricos*, como uma impressora, constituem o *hardware*. O que nos é útil é um conjunto *software + hardware*. Um conjunto deste tipo é chamado de um *sistema de computação*. De agora em diante, os nossos processadores serão *sistemas de computação*. Isto é, queremos escrever programas que sejam executado por um *sistema de computação*.

Como foi dito acima, o desenvolvimento de um programa que gerencie o dia a dia comercial de uma farmácia requer um compilador (ou um interpretador) que o traduza para a linguagem de máquina. Antigamente, as empresas que desenvolviam compiladores desenvolviam apenas estes programas, de tal sorte que o programador necessitava utilizar um processador de texto à parte para edição do programa fonte. Atualmente, os compiladores são integrados num sistema de computação que contém, entre outros:

1. *Processador de texto*, para a digitação dos programas fontes;
2. *Depurador*, que permite que o programa seja executado comando a comando, o que facilita a descoberta de erros de lógica;
3. *Help*, que descreve as sintaxes e as semânticas de todas as instruções da linguagem;
4. *Linker*, que permite que um programa utilize outros programas.

Rigorosamente falando, um sistema constituído de um compilador e os *softwares* listados acima deveria ser chamado de *ambiente de programação*; é mais comum, entretanto, chamá-lo, simplesmente, de *compilador*.

Os ambientes de programação que utilizamos para desenvolver os programas deste livro foram o compilador Turbo C, versão 2.01, e Turbo C++, versão 3.0, ambos desenvolvidos pela *Borland International, Inc.*, o primeiro em 1988 e o segundo em 1992. Como se pode ver, são sistemas desenvolvidos há bastante tempo (as coisas em computação andam muito mais rápido), já estando disponíveis gratuitamente na *internet.* Estaremos, portanto, utilizando um compilador "puro C" e um compilador C++, que é *up grade* da linguagem C para a *programação orientada a objeto*, paradigma que não está no escopo deste livro.

1.11 Exercícios propostos

1. Três índios, conduzindo três brancos, precisam atravessar um rio dispondo para tal de um barco cuja capacidade é de apenas duas pessoas. Por questões de segurança, os índios não querem ficar em minoria, em nenhum momento e em nenhuma das margens. Escreva um algoritmo que oriente os índios para realizarem a travessia nas condições fixadas. (Cabe observar que, usualmente, este exercício é enunciado envolvendo três jesuítas e três canibais. A alteração feita é uma modesta contribuição para o resgate da verdadeira história dos índios).

2. O jogo conhecido como *Torre de Hanói* consiste de três torres chamadas *origem*, *destino* e *auxiliar* e um conjunto de n discos de diâmetros diferentes, colocados na torre *origem* na ordem decrescente dos seus diâmetros. O objetivo do jogo é, movendo um único disco de cada vez e não podendo colocar um disco sobre outro de diâmetro menor, transportar todos os discos para torre *destino*, podendo usar a torre *auxiliar* como passagem intermediária dos discos. Escreva algoritmos para este jogo nos casos n = 2 e n = 3.

3. Imagine que se disponha de três esferas numeradas 1, 2 e 3 iguais na forma, duas delas com pesos iguais e diferentes do peso da outra. Escreva um algoritmo que, com duas pesagens numa balança de dois pratos, determine a esfera de peso diferente e a relação entre seu peso e o peso das esferas de pesos iguais.

4. A *média geométrica* de *n* números positivos é a raiz n-ésima do produto destes números. Supondo que o processador é capaz de calcular raízes n-ésimas, escreva um algoritmo para determinar a média geométrica de *n* números dados.

5. Sabendo que o dia 01/01/1900 foi uma segunda-feira, escreva um algoritmo que determine o dia da semana correspondente a uma data, posterior a 01/01/1900, dada. Por exemplo, se a data dada for 23/01/1900, o algoritmo deve fornecer como resposta terça-feira.

6. O show de uma banda de rock, que será realizado na margem de um rio, deve começar exatamente às 21 h. Atrasados, às 20 h 43 min, os quatro integrantes da banda estão na outra margem do rio e necessitam, para chegar ao palco, atravessar uma ponte. Há somente uma lanterna e só podem passar uma ou duas pessoas juntas pela ponte, e sempre com a lanterna. Cada integrante possui um tempo diferente para atravessar a ponte: o vocal leva 10 minutos, o guitarrista 5 minutos, o baixista 2 minutos e o baterista 1 minuto. Evidentemente, quando dois atravessam juntos, o tempo necessário é o do mais lento. Escreva um algoritmo que permita que a banda atravesse a ponte de modo que o show comece na hora marcada.

7. Resolva a questão 3 para o caso de oito esferas, com três pesagens.

8. Escreva um algoritmo para determinar o resto de uma divisão inteira utilizando uma máquina de calcular que efetue apenas as quatro operações: adição, subtração, multiplicação e divisão.

Observação

Propostas de soluções dos exercícios propostos podem ser solicitadas através de mensagem eletrônica para jaime@ccen.ufal.br com assunto RESPOSTAS LIVRO C, anexando o formulário abaixo devidamente preenchido.

Nome	Categoria[1]	Instituição[2]	Curso[2]	Cidade/Estado

[1]Categoria: docente, estudante, autodidata
[2]Se docente ou estudante

2. Introdução à Linguagem C

2.1 Variáveis simples

O que é uma variável

No capítulo 1 foi dito que uma das unidades básicas de um computador é a *memória*, cuja finalidade é armazenar dados e informações que serão manipulados pela *unidade central de processamento.* Naquele capítulo também foi dito que os programas para serem executados devem estar armazenados na memória. Ou seja, a memória armazena programas que serão executados e dados que estes programas vão manipular. Naturalmente, os dados que o programa vai manipular podem ser *dados de entrada* ou dados gerados pela execução do programa.

Para que possa armazenar dados e informações, a memória é dividida em partes, chamadas *posições de memória*. O sistema operacional que gerencia o sistema de computação pode acessar cada uma destas posições para armazenar tais dados. Para que o acesso às posições de memória seja possível, a cada uma delas está associada uma sequência de bit's, chamada *endereço* da posição de memória. Como uma sequência de bit's corresponde a um número inteiro escrito no sistema binário, cada endereço pode ser visto como um inteiro escrito no sistema decimal. Assim temos posições de memória de endereço 1209 ou 2114, por exemplo.

Uma *variável simples* (ou simplesmente *variável*) é uma posição de memória cujo conteúdo pode ser modificado durante a execução de um programa. A referência a uma variável no programa é feita através do seu *identificador*; os valores que podem ser nela armazenados dependem do seu *tipo de dado*.

O identificador

O *identificador* é uma sequência de letras, dígitos e caractere para sublinhamento escolhida pelo programador e (como foi dito acima) será utilizado no programa para se fazer referência àquela variável (o primeiro caractere do identificador não pode ser um dígito). Como um programa deve ser legível por outros programadores (e pelo próprio programador), é uma boa prática se escolher um identificador de uma variável que tenha alguma relação com a sua finalidade. Se uma variável deve armazenar uma soma, um identificador muito bom para ela será *Soma*. Se uma variável vai receber números, ela poderia ser identificada por *Num* ou por *Numero*. Os compiladores da linguagem C fazem distinção entre letras maiusculas e minúsculas e, portanto, *Numero* e *numero* são dois identificadores diferentes. Manteremos, de um modo geral, a seguinte convenção ao longo do texto: quando um identificador possuir mais de um caractere, iniciá-lo-emos por letra maiuscula e quando o identificador tiver um único caractere, utilizaremos letra minúscula.

Como veremos ao longo do livro, a linguagem C fixa alguns identificadores para a *sintaxe* de suas instruções. Estes identificadores não podem ser utilizados nos programas, sendo conhecidos por *palavras reservadas*. Algumas das palavras reservadas em C são:

Tabela 2 Palavras reservadas da linguagem C

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	Signed	void
default	goto	sizeof	volatile
do	if	static	while

O tipo de dado

O *tipo de dado* associado a uma variável é o conjunto dos valores que podem ser nela armazenados. A linguagem C dispõe dos tipos de dados discriminados na tabela a seguir.

Tabela 3 Tipos de dados da Linguagem C

Denominação	Número de Bytes	Conjunto de valores
char	1	caracteres codificados no código ASCII
int	2	números inteiros de –32768 a 32767
long ou long int	4	números inteiros de –65536 a 65535
float	4	números reais de $-3{,}4x10^{38}$ a $-3{,}4x10^{-38}$ e $3{,}4x10^{-38}$ a $3{,}4x10^{38}$
double	8	números reais de $-1{,}7x10^{308}$ a $-1{,}7x10^{-308}$ e $1{,}7x10^{-308}$ a $1{,}7x10^{308}$
void	0	conjunto vazio

A utilização *void* será melhor explicada no capítulo 5, quando estudarmos *funções.*

Uma observação importante é que os tipos *float* e *double*, rigorosamente falando, não armazenam números reais e sim números de um *sistema de ponto flutuante*, que não contém todos os reais entre dois números reais dados. O estudo de sistemas de ponto flutuante foge ao escopo deste livro e é feito, normalmente, em disciplinas do tipo *Organização e Arquitetura de Computadores* e *Cálculo Numérico*.

Vale lembrar que, de um modo geral, um byte contém oito bit's e cabe ressaltar que, em algumas situações, é importante se conhecer a quantidade necessária de bytes para uma variável de um determinado tipo.

Declaração de variáveis

Para que o sistema de computação possa reservar as posições de memória que serão utilizadas pelo programa, associar identificadores aos endereços destas posições de memória e definir a quantidade de bytes de cada posição de acordo com o tipo de dado pretendido, um programa escrito em C deve conter a *declaração de variáveis*, feita através da seguinte sintaxe:

Tipo de dado Lista de identificadores;

Por exemplo, um programa para determinar a média de uma relação de números dados pode ter a seguinte declaração:

```
int Quant;
float Num, Soma, Media;
```

A ideia é que *Quant* seja utilizada para armazenar a quantidade de números; *Num* para armazenar os números (um de cada vez); *Soma* para armazenar a soma dos números; e *Media* para armazenar a média procurada.

Nas seções 2.7 e 2.9 veremos as instruções em C para o armazenamento em variáveis de dados de entrada e de dados gerados pela execução do algoritmo. Um valor armazenado em uma variável é comumente referido como sendo o *conteúdo* da variável ou o *valor* da variável. Também é comum se referir ao identificador da variável como sendo a própria variável.

2.2 Constantes

Como uma variável, uma *constante* também é uma posição de memória à qual devem ser associados um *identificador* e um *tipo de dado*. O que caracteriza uma *constante* (e daí sua denominação, emprestada da matemática) é o fato de que o conteúdo de uma constante não pode ser modificado durante a execução do programa. Este conteúdo é fixado quando da declaração da *constante* o que deve ser feito de acordo com a seguinte sintaxe:

const Tipo de Dado Identificador = Valor;

Por exemplo, um programa para processar cálculos químicos poderia ter uma declaração do tipo

```
const  float NumAvogadro = 6.023E+23;
```

onde a expressão 6.023E+23 é a forma que os compiladores C utilizam para representar os valores do tipo *float* na *notação científica*, ou seja 6.023E+23 = 6.023 x 10^{23}.

Um programa para cálculos de áreas de figuras planas, perímetros de polígonos inscritos em circunferências, etc., poderia ter uma declaração do tipo

```
const float Pi = 3.1416;
```

Esta declaração é desnecessária quando o sistema utilizado é o Turbo C++ 3.0, pois esse sistema disponibiliza uma *constante pré-definida*, identificada por M_PI, que pode ser utilizada em qualquer parte do programa e cujo valor é uma aproximação do número irracional π.

2.3 Expressões aritméticas

Como era de se esperar, os compiladores da linguagem C são capazes de avaliar expressões aritméticas que envolvam as operações binárias de multiplicação, divisão, soma e subtração e a operação unária de *troca de sinal*. Para isto são usados os seguintes *operadores aritméticos binários*:

Tabela 4 Operadores aritméticos

Operador	Operação
+	adição
-	subtração
*	multiplicação
/	divisão

e o *operador aritmético unário* (**-**) para a troca de sinal. Esses operadores atuam com operandos do tipo *int* ou do tipo *float*. Se um dos operandos for do tipo *float* o resultado da operação será do tipo *float*; se os dois operandos forem do tipo *int* o resultado é também do tipo *int*. No caso do operador de divisão /, se os dois operandos forem do tipo *int* o resultado da operação é do tipo *int* e igual ao *quociente* da divisão do primeiro operando pelo segundo. Por exemplo, o resultado de 30/4 é 7. Se quisermos a divisão decimal teremos de escrever 30.0 / 7.0 ou 30.0 / 7 ou 30 / 7.0. Ou seja cada uma destas divisões é igual a 7.5 e este valor, tal como ele é, pode ser armazenado numa variável do tipo *float*. O que acontece é que no armazenamento de um valor do tipo *float* numa variável do tipo *int* a parte decimal do valor é desprezada, só sendo armazenada a parte inteira do número.

Uma expressão que envolva diversas operações é avaliada de acordo com as regras de prioridade da matemática: em primeiro lugar é realizada a operação troca de sinal, em seguida são realizadas as multiplicações e divisões e, finalmente, as somas e subtrações. Por exemplo, a expressão 8 + 2*-3 é avaliada como 8 + (-6) = 2. Naturalmente, a prioridade pode ser alterada com a utilização de parênteses: a expressão (8 + 2)*-3 resulta em 10*(-3) = -30. Embora, o sistema não exija, vamos sempre utilizar parênteses para separar o operador unário para troca de sinal de algum operador binário. Assim, 8 + 2*-3 será indicada por 8 + 2*(-3). Uma expressão não parentesada contendo operadores de mesma prioridade é avaliada da esquerda para direita. Por exemplo, 10/2*3 é igual a (10/2)*3 = 5*3 = 15.

Operandos podem ser conteúdos de variáveis. Neste caso, o operando é indicado pelo *identificador* da variável (é para isto que serve o identificador, para se fazer referência aos valores que na variável estão armazenados).

Além dos operadores aritméticos usuais, os compiladores C disponibilizam o *operador módulo*, indicado por %, que calcula o resto da divisão do primeiro operando pelo segundo. Por exemplo, 30 % 4 = 2 e 5 % 7 = 5. Este operador atua apenas em operandos do tipo *int*, resultando um valor deste mesmo tipo. Por exemplo, se S é uma variável do tipo *float*, a expressão S % 5 gerará um erro de compilação. Uma expressão do tipo 30.0 % 7 também gerará erro de compilação, pelo fato de que um dos operandos não é inteiro. Este erro é indicado pelo sistema pela mensagem *Illegal use of floating point in function ...* (Uso ilegal de tipo *float* na função ...), onde as reticências estão substituindo o *identificador da função*, como será discutido posteriormente.

2.4 Relações

Os ambientes que implementam a linguagem C efetuam *comparações* entre valores numéricos, realizadas no sentido usual da matemática. Essas comparações são chamadas *relações* e são obtidas através dos *operadores relacionais* > (*maior do que*), >= (*maior do que ou igual a*), < (*menor do que*), <= (*menor do que ou igual a*), == (*igual*) e != (*diferente*).

O resultado da avaliação de uma relação é 1 (um), se a relação for matematicamente verdadeira, ou 0 (zero), se a relação for matematicamente falsa. Assim, 3 > 5 resulta no valor 0 (zero), enquanto que 7 <= 7 resulta no valor 1 (um). Sendo um valor 1 (um) ou 0 (zero), o resultado da avaliação de uma relação pode ser armazenado numa variável do tipo *int*.

Os operandos de uma relação podem ser expressões aritméticas. Nestes casos, as expressões aritméticas são avaliadas em primeiro lugar para, em seguida, ser avaliada a relação. Por exemplo, a relação 3*4 - 5 < 2*3 - 4 resulta no valor 0 (zero), pois 3*4 - 5 = 7 e 2*3 - 4 = 2. Isto significa que os operadores relacionais têm prioridade mais baixa que os aritméticos.

2.5 Expressões lógicas

Os compiladores C também avaliam *expressões lógicas* obtidas através da aplicação dos *operadores lógicos binários* &&, || e ^ a duas relações ou da aplicação do *operador lógico unário* ! a uma relação.

Se R_1 e R_2 são duas relações, a avaliação da aplicação dos operadores lógicos binários, de acordo com os valores de R_1 e R_2, são dados na tabela abaixo.

Tabela 5 Avaliação de expressões lógicas

<table>
<tr><td>R₁</td><td>R₂</td><td>(R₁)&&(R₂)</td><td>(R₁)||(R₂)</td><td>(R₁) ^ (R₂)</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
</table>

Ou seja, uma expressão lógica do tipo (R_1)&&(R_2) só recebe o valor 1 (um) se os valores de R_1 e de R_2 forem iguais a 1 (um); uma expressão lógica do tipo (R_1)||(R_2) só recebe o valor 0 (zero) se os valores de R_1 e de R_2 forem iguais a 0 (zero); uma expressão lógica do tipo (R1) ^ (R2) só recebe o valor 1 (um) se apenas um dos valores de R1 e R2 for igual a 1. O leitor já deve ter percebido que o operador && age como o conectivo *e* da nossa linguagem; o operador || atua como o nosso *e/ou* e o operador ^ como o conectivo *ou*.

A aplicação do operador unário ! simplesmente inverte o valor original da relação:

Tabela 6 Operador unário !

R_1	$!R_1$
1	0
0	1

Considerando que os operadores &&, || e ^ possuem o mesmo grau de prioridade, se uma expressão não parentesada possuir mais de uma relação, ela será avaliada da esquerda para direita. O operador unário ! tem prioridade em relação aos operadores binários. Assim, ! (5 > 3) || (5 < 3) tem valor 0 (zero), pois ! (5 > 3) é uma relação falsa e 5 < 3 também é. Considerando que os operadores lógicos têm prioridade mais baixa que os operadores relacionais, os parênteses nas expressões acima são desnecessários; porém entendemos que a colocação deles facilita a leitura da expressão.

Os sistemas C 2.01 e C++ 3.0 também disponibilizam os operadores lógicos & e | cujas aplicações são idênticas às aplicações de && e ||, respectivamente. A diferença entre & e &&, por exemplo, é a seguinte: se em (R1) && (R2) o valor de R1 for 0 (zero) o valor R2 não é mais avaliado, enquanto que em (R1) & (R2) o valor de R2 é avaliado, independentemente do valor de R1.

2.6 Estrutura de um programa em C

Estamos tentando aprender a escrever programas na linguagem C. Já vimos que se o programa necessitar manipular variáveis, estas devem ser declaradas. Veremos no capítulo 5 que um programa pode conter *funções*. Na verdade, veremos que um programa em C é um conjunto de funções definidas pelo programador, funções que utilizarão outras funções definidas pelo programador e algumas funções oferecidas pelo sistema (as funções oferecidas pelo sistema são chamadas *funções de biblioteca* ou *funções pré-definidas*). Veremos no citado capítulo que uma função deve ser definida com a seguinte estrutura.

```
Tipo de Dado Identificador da função(Lista de parâmetros)
{
Declaração de variáveis
Sequência de instruções
}
```

onde o significado de *Lista de parâmetros* será explicado no capítulo já referido e a *Sequência de instruções* contém *comandos*, *ativações de funções pré-definidas* e *ativações de funções definidas pelo usuário no próprio programa ou em outros programas*.

Todo programa em C deve conter uma função identificada por *main* (cuja tradução é *principal*), com lista de parâmetros vazia e tipo de dado não obrigatório. Esta será sempre a primeira função do programa a ser executada. Desta forma, o menor programa em C é

```
main()
{
}
```

Este programa pode ser executado, mas nada realiza, devido ao fato de que ele não contém nenhuma instrução. Observe que todo o *corpo* da função deve estar disposto entre chaves. As chaves são utilizadas em outros pontos de um programa e são chamadas *delimitadores*. Os delimitadores, o identificador *main* e os parênteses, dispostos ao lado do identificador *main*, são os únicos elementos obrigatórios de um programa. Como os compiladores C ignoram espaços em branco, caracteres de tabulação e caracteres de mudança de linha, não existe um estilo obrigatório de se editar programas em C. Por exemplo, o programa acima poderia ser escrito de uma das seguintes maneiras.

```
  main
(
)
                  {
                  }
```

ou

```
main(){}
```

ou

```
main(
){}
```

Evidentemente, um programador em C deve procurar escrever seus programas num estilo que ofereça uma boa legibilidade, o que vai facilitar a sua compreensão por outra pessoa e a descoberta de possíveis erros de lógica.

2.7 Entrada dos dados de entrada

A função *scanf()*

A maioria dos programas manipula dados que são fornecidos pelo usuário durante a execução do

programa. Estes dados constituem a *entrada do programa* e, naturalmente, devem ser armazenados em variáveis. Por exemplo, um programa para determinação das raízes de uma equação do segundo grau deve receber como entrada os valores dos três coeficientes da equação: são estes valores que identificam a equação.

De um modo geral, os compiladores C permitem a recepção de dados de entrada (e o consequente armazenamento em variáveis) através da função de biblioteca *scanf()* que possui a seguinte sintaxe:

scanf(Expressão de controle, Lista de variáveis);

Aí, *Expressão de controle* deve ser escrita entre aspas e contém os *códigos de conversão* que indicam como o sistema deve armazenar os dados digitados no teclado e caracteres que o usuário deve digitar separando a digitação destes dados. Na *Lista de variáveis* as variáveis são separadas por vírgulas e cada uma delas deve ser precedida do *operador de endereço* &. Este operador indica o endereço da posição de memória definida para a variável identificada na lista de variáveis.

Quando da execução desta função, a *janela de edição* é substituída pela *janela do usuário* e o sistema fica aguardando que o usuário digite um número de valores igual ao número de variáveis da lista de variáveis (à medida que são digitados, os valores aparecem na tela do usuário). A conclusão da entrada dos dados é feita com a digitação da tecla <enter> e quando isto é feito, o sistema armazena os dados digitados na variável respectiva (no sentido da ordem da colocação da variável na lista e da digitação do valor), de acordo com o *código de conversão*.

Por exemplo,

```
#include <stdio.h>
main()
{
int Anos
scanf("%d", &Anos)
}
```

é um programa em C que armazena um valor inteiro digitado no teclado na variável *Anos* (ou seja, para nada serve, pois o inteiro armazenado naquela posição de memória "esvanece" quando a execução do programa é encerrada).

Quando se vai dar entrada em mais de um dado através de uma mesma ativação da função *scanf()*, pode-se fixar caracteres que deverão ser digitados quando da entrada dos dados. Por exemplo, se pretendemos dar entrada numa data, podemos ter o seguinte trecho de programa:

```
#include <stdio.h>
main()
{
int Dia, Mes, Ano;
scanf("%d/%d/%d", &Dia, &Mes, &Ano);
. . .
}
```

e a data pretendida deve ser digitada no formato dd/mm/aaaa, devendo o usuário digitar as barras entre as digitações do dia e do mês e deste e do ano. A conclusão se dá com a digitação da tecla <enter>, como se estivesse digitando um único dado. A digitação das barras é necessária pelo fato de que elas estão separando os códigos de formatação na expressão de controle. Se após a digitação do valor da variável *Dia* for acionada a tecla <enter>, a execução da função é encerrada e os valores de *Mes* e *Ano* não podem ser digitados.

Se não há nenhum caractere separando os códigos de formatação, a digitação dos dados pode ser intercalada pela digitação da tecla <enter> ou da <barra de espaços>. É o caso, por exemplo, do trecho de programa

```
main()
{
int i;
char c;
scanf("%c %d", &c, &i);
. . .
}
```

onde as digitações do caractere que se pretende armazenar na variável *c* e do inteiro a ser armazenado em *i* devem ser separadas pelo acionamento da tecla <enter> ou da <barra de espaço>. É necessário notar que a digitação de um valor de um tipo diferente do tipo da variável não provoca erro de execução, mas, evidentemente, pode provocar erro de lógica do programa. Por exemplo, se na execução do comando *scanf("%c %d", &c, &i);* digitarmos w<enter>5.9<enter>, o caractere *w* é armazenado na variável *c* e o inteiro 5 é armazenado na variável *i*. Portanto, se o dado de entrada fosse realmente 5.9, o resultado do processamento poderia ser fornecido com erros. Se o dado de entrada **poderia ser** 5.9, a variável para seu armazenamento deveria ter sido definida com *float*.

Os códigos de conversão e a instrução #include <stdio.h>

Os *códigos de conversão* de acordo com o tipo de dado da variável onde os valores digitados serão armazenados são apresentados na tabela a seguir.

Tabela 7 Códigos de conversão da função scanf()

Código	Elemento armazenado
%c	um único caractere
%d ou %i	um inteiro do sistema decimal
%o	um inteiro do sistema octal
%x	um inteiro do sistema hexadecimal
%ld	um valor do tipo *long*
%e	um número na notação científica
%f	um número em ponto flutuante
%s	uma cadeia de caracteres

A instrução *#include <stdio.h>* que precede a função *main()* é necessária pelos seguintes fatos. Como dissemos acima, para se definir uma função é necessário fixar o tipo de dado que ela *retorna*, o *identificador da função* e a *lista de parâmetros*, com seus identificadores e seus tipos de dados; este conjunto de elementos é chamado *protótipo da função*. Para que a função *main()* ative uma outra função (seja uma função definida pelo usuário ou uma *função de biblioteca*), o seu protótipo deve ser definido antes ou no interior da função *main()*.

Os protótipos das funções do sistema encontram-se reunidos, de acordo com objetivos semelhantes, em arquivos chamados *arquivos de cabeçalhos* (*header files*) (o *cabeçalho* de uma função inclui o seu protótipo, as variáveis declaradas dentro da função e outras declarações e definições que não são instruções propriamente ditas). A instrução *#include <stdio.h>* "anexa" à função *main()* os protótipos das funções de biblioteca que executam ações padrões de entrada e de saída (*stdio* vem de *standard input output*, entrada e saída padrão e *h* é a extensão padrão dos arquivos de cabeçalhos). A não inclusão de um *include* provoca erro de compilação no sistema C++ 3.01. Isto não acontece no C 2.01, porém, há casos em que esta não inclusão gera erros de lógica (a entrada de dados não é feita do modo que se esperava).

2.8 Saída de dados

A função printf()

A exibição dos resultados do processamento e de mensagens é feita através da função pré-definida *printf()*, cujo protótipo está contido também no arquivo *stdio.h*. Sua sintaxe é a seguinte:

printf(Expressão de controle, Lista de argumentos);

onde *Expressão de controle* contém *mensagens* que se pretende que sejam exibidas, *códigos de formatação* (idênticos aos códigos de conversão da função *scanf()*) que indicam como o conteúdo de uma variável deve ser exibido e *códigos especiais* para a exibição de alguns caracteres especiais e realização de ações que permitam formatar a saída do programa. A *Lista de argumentos* pode conter identificadores de variáveis,

expressões aritméticas ou lógicas e valores constantes.

No primeiro caso, o conteúdo da variável é exibido; no segundo caso, a expressão é avaliada e o seu resultado é exibido; no terceiro caso o valor constante é exibido. A ordem de exibição dos conteúdos de variáveis, dos resultados das expressões e dos valores constantes relacionados na lista de argumentos é dada pela ordem em que estes elementos estão listados; a posição dentro da mensagem contida na expressão de controle é fixada pela posição do código de formatação respectivo. Quando, na expressão de controle, um código de formatação é encontrado o conteúdo da variável, o resultado da expressão ou o valor constante respectivo (no sentido da ordem da colocação da variável na lista e da colocação do código de formatação na expressão de controle) é exibido.

Por exemplo, a função *printf()* no programa abaixo contém uma expressão de controle que não possui códigos de formatação. Isto significa que apenas a mensagem será exibida. Assim, o programa

```
#include <stdio.h>
main()
{
printf("Estou aprendendo a programar em C");
}
```

é um programa em C que faz com que seja exibida na tela a mensagem *Estou aprendendo a programar em C*.

Já o programa abaixo, contém uma função *printf()* que possui quatro caracteres de controle

```
#include <stdio.h>
main()
{
float a, b, c;
scanf("%f %f %f", &a, &b, &c);
printf("%f , %f e %f %f", a, b , c, (a + b + c)/3);
}
```

Quando da execução deste programa, o sistema, para execução da função *scanf()*, aguarda que sejam digitados três valores numéricos. Quando isto é feito, o sistema armazena estes três valores nas variáveis *a*, *b* e *c*, respectivamente. Na execução do último comando, o sistema exibe os valores armazenados nas variáveis *a*, *b* e *c*, em seguida avalia a expressão *(a + b + c)/3* e exibe o seu valor na tela. Assim, o programa fornece a média aritmética de três números dados.

Como um outro exemplo e considerando que o resultado de uma expressão lógica é um inteiro, o programa

```
#include <stdio.h>
main()
{
printf("%d", 5 > 3);
}
```

exibe na tela o valor 1, pois a relação 5 > 3 é verdadeira.

Nos dois exemplos anteriores, utilizamos expressões, uma aritmética e uma lógica, como argumentos de uma função *printf()*. No nosso entendimento, não é uma boa prática de programação se utilizar expressões como argumentos de uma função *printf()*. Se o valor de uma expressão é útil para alguma coisa, ele deve ser armazenado em alguma variável (veremos isto na próxima seção) e esta deve ser utilizada para o fornecimento de resultados.

Facilitando a execução de um programa

A possibilidade de que mensagens possam ser exibidas permite que o próprio programa facilite a sua execução e que torne compreensíveis os resultados fornecidos. Da forma em que está escrito acima, a execução do programa que fornece a média de três números dados é dificultada pelo fato de que a execução da função *scanf()* faz com que o sistema aguarde a digitação dos números pretendidos (o *cursor* fica simplesmente piscando na tela do usuário) e o usuário pode não saber o que está se passando. Além disto, a

execução da função *printf()* exibe apenas o resultado da expressão, sem indicação a que aquele valor se refere. Sem dúvida, o programa referido ficaria muito melhor da seguinte forma.

```
#include <stdio.h>
main()
{
float a, b, c;
printf("Digite três números");
scanf("%f %f %f", &a, &b, &c);
printf("A media dos numeros %f , %f e %f é igual a %f", a, b, c, (a + b + c)/3);
}
```

A exibição de uma mensagem pode ser também obtida através da função *puts()*, cujo protótipo está no arquivo *stdio.h*. Por exemplo, o comando *printf("Digite três números")* pode ser substituído pelo comando *puts("Digite três números")*.

Fixando o número de casas decimais

O padrão utilizado pela maioria dos compiladores C é exibir os números de *ponto flutuante* com seis casas decimais. O número de casas decimais com as quais os números de ponto flutuante serão exibidos pode ser alterado pelo programa. Para isso deve-se acrescentar **.n** ao código de formatação da saída, sendo *n* o número de casas decimais pretendido. Por exemplo, se o programa que determina a média de três números fosse executado para a entrada 6.2, 8.45 e 7 seria exibido na tela o seguinte resultado

A media dos numeros 6.200000, 8.550000 e 7.000000 é igual a 7.250000

Se o comando de saída do programa fosse

```
printf("A media dos numeros %.2f , %.2f e %.2f é igual a %.1f", a, b, c, (a + b + c)/3);
```

a saída seria

A media dos numeros 6.20, 8.55 e 7.00 é igual a 7.3

Observe que a média dos números dados, de fato, é igual a 7.26. Como o código da formatação da saída da média foi *%.1f*, ela foi exibida com uma casa decimal e o sistema efetua os arredondamentos necessários. Observe também a utilização do ponto (e não da vírgula) como separador das partes inteiras e fracionárias. Isto é sempre necessário quando o ambiente de programação que se está utilizando foi desenvolvido nos Estados Unidos, o que é o mais frequente.

Alinhando a saída

O programa pode fixar a coluna da tela a partir da qual o conteúdo de uma variável, ou o valor de uma constante ou o valor de uma expressão será exibido. Isto é obtido acrescentado-se um inteiro **m** ao código de formatação. Neste caso, *m* indicará o número de colunas que serão utilizadas para exibição do conteúdo da variável ou do valor da constante. Por exemplo, levando-se em conta que a frase "Estou aprendendo a programar" contém vinte e oito caracteres, o programa abaixo

```
#include <stdio.h>
main()
{
printf("%38s",  "Estou aprendendo a programar");
}
```

exibe na tela a frase referida a partir da décima coluna.

Observe que este programa também exemplifica a utilização de uma constante (no caso, uma cadeia de caracteres) como um argumento da função *printf()*. Observe também que referências a constantes do tipo cadeia de caracteres devem ser feitas com a cadeia escrita entre aspas. As aspas distinguem para o sistema

uma cadeia de caracteres constante de um identificador de variável. O mesmo efeito poderia ser obtido com o programa

```
#include <stdio.h>
main()
{
printf("          Estou aprendendo a programar");
}
```

onde existem dez espaços em branco entre o abre aspas e a letra E.

Para se fixar a coluna a partir da qual e o número de casa decimais com que um número de ponto flutuante será exibido, deve-se utilizar dois parâmetros separados por um ponto. Por exemplo, considerando que se exibirmos o número 3.1416 com duas casas decimais ele ficará com quatro caracteres, o programa

```
#include <stdio.h>
main()
{
printf("%14.2f", 3.1416);
}
```

exibirá na tela 3.14 a partir da décima coluna.

O recurso da exibição de valores utilizando um número pré-definido de colunas pode ser utilizado para alinhar à direita a saída do programa. Por exemplo, se os conteúdos das variáveis do tipo *float* x, y e z são 103.45, 5.3678 e 45.0, a sequência de comandos

```
printf("%13.2f", x);
printf("%13.2f", y);
printf("%13.2f", z);
```

exibe na tela

```
103.45
  5.37
 45.00
```

Vale observar que é possível obter o mesmo efeito com uma única ativação (ou *chamada*) da função *printf()*, como veremos a seguir:

```
printf("%13.2f \n %13.2f \n %13.2f ", x, y, z);
```

Exibindo números "como caracteres" e vice-versa

Uma variável do tipo *char* armazena os códigos ASCII dos caracteres suportados pelo sistema. Como este armazenamento é feito através de cadeias de bit's, na verdade, o que é armazenado são números inteiros compreendidos entre –128 e 127. Aos números de 0 a 127 correspondem os caracteres de código ASCII iguais ao próprio número e aos números de -1 a -128 correspondem os caracteres de código ASCII iguais aos números situados no intervalo de 128 a 255, respectivamente.

O código de formatação da saída é que indicará a forma como o conteúdo de uma variável do tipo *char* será exibido. Se *c* é uma variável do tipo *char*, pode-se associar a sua saída com os códigos %d, %o, %x, %c. No primeiro caso o número armazenado em *c* é exibido; no segundo caso este número será exibido no sistema octal; no terceiro, o número será exibido no sistema hexadecimal e no último caso será exibido o caractere como comentado acima. Por exemplo, se o conteúdo da variável *char c* é 67, o comando

```
printf("%c  %d  %o  %x", c, c, c, c);
```

exibirá na tela

```
C 67 103 43
```

A razão é disto é que 67 é o código ASCII de C no sistema decimal, 103 é este código no sistema octal e 43 é o código ASCII de C no sistema hexadecimal.

Quando um argumento é uma constante, ele será exibido de forma que depende do código de

formatação. Quando a constante é uma cadeia de caracteres, não há muito o que fazer: a execução do comando

```
printf("Este comando exibirá  a palavra %s", "paz");
```

exibirá na tela a frase

Este comando exibirá a palavra paz

da mesma maneira que o comando

```
printf("Este comando exibirá  a palavra paz");
```

que é muito mais simples.

Porém, quando a constante é um caractere ou um número inteiro, o que será exibido depende do código de formatação. Por exemplo, o comando

```
printf("%c", 'A');
```

exibe o caractere A, enquanto que o comando

```
printf("%d", 'A');
```

exibirá o número 65. Por sua vez, o comando

```
printf("%d", 65);
```

exibe o número 65, enquanto que o comando

```
printf("%c", 65);
```

exibe o caractere A.

Observe que referências a constantes caracteres é feita com o caractere escrito entre apóstrofos, enquanto que referências a cadeias de caracteres são feitas com as cadeias escritas entre aspas, como já foi observado antes.

Os códigos especiais

De acordo com a tabela abaixo, os *códigos especiais* permitem a exibição de alguns caracteres, como %, \, dentro de uma mensagem e a realização de ações que permitem a formatação da saída de modo que esta seja elegante e agradável para o usuário.

Tabela 9 Códigos especiais da função printf()

Código	Ação
\n	leva o cursor para a próxima linha
\t	executa uma tabulação
\b	executa um retrocesso
\f	leva o cursor para a próxima página
\a	emite um sinal sonoro (*beep*)
\"	exibe o caractere "
\\	exibe o caractere \
\%	exibe o caractere %

Uma observação interessante é que o código **\a** pode ser obtido através do caractere de código ASCII igual a 7. Assim, a execução dos comandos *printf*("\a"); e *printf*("%c", 7); realizam a mesma ação de emissão de um sinal sonoro.

A utilização do código **\n** permite que a exibição de constantes ou de conteúdos de variáveis através da função *printf()* possa ser feita em linhas diferentes. No exemplo dado na seção anterior sobre o alinhamento da saída dos dados, a saída poderia ser feita com uma única chamada da função *printf()*. Repetindo o referido exemplo, se os conteúdos das variáveis do tipo *float* x, y e z são 103.45, 5.3678 e 45.0, o comando

```
printf("%13.2f"/n %13.2f \n %13.2f", x, y, z);
```

exibe na tela

103.45
  5.37
45.00

2.9 Comando de atribuição

Armazenando dados gerados pelo programa

A seção 2.7 apresentou o comando que permite que se armazene em variáveis a entrada do programa. Agora veremos como armazenar dados gerados durante a execução de um programa. Considere um programa para o cálculo da média de uma relação de números. Naturalmente, a quantidade de números da relação (se não foi fornecida a priori) deve ser de alguma forma determinada e armazenada em alguma variável para que possa ser utilizada no cálculo final da média pretendida.

O armazenamento de dados gerados pelo próprio programa, alterações no conteúdo de variáveis e determinações de resultados finais de um processamento são realizados através do *comando de atribuição*, que deve ser escrito com a seguinte sintaxe.

Identificador de variável = expressão;

A expressão do segundo membro pode se resumir a um valor constante pertencente ao tipo de dado da variável do primeiro membro, caso em que o valor é armazenado naquela variável. Se não for este o caso, a expressão é avaliada e, se for do mesmo tipo da variável do primeiro membro, o resultado é armazenado na variável.

A expressão do segundo membro pode envolver a própria variável do primeiro membro. Neste caso, o conteúdo anterior da variável será utilizado para a avaliação da expressão e será substituído pelo valor desta expressão. Por exemplo, se *i* é uma variável do tipo *int* ou do tipo *float* o comando i = i + 1; faz com que o seu conteúdo seja incrementado de uma unidade.

Veremos ao longo do livro que comandos do tipo i = i + 1; aparecem com muita frequência. A linguagem C oferece uma forma simplificada de escrever este comando: i++;. Esta sintaxe se tornou tão característica da linguagem C que sua "ampliação" para incorporar recursos de programação orientada a objetos foi denominada C++ (de forma semelhante, o comando i = i – 1 pode ser escrito i--;). O incremento de uma variável de uma unidade também pode ser obtido através do comando ++i e estas expressões podem figurar em expressões aritméticas. A diferença entre i++ e ++i pode ser entendida no seguinte exemplo. A sequência de comandos

```
i = 2;
j = i++;
k = ++i;
```

realiza as seguintes ações:

i = 2, armazena em *i* o valor 2;
j = i++, armazena em *j* o valor 2 e armazena em *i* o valor 3 (incrementa o valor de *i*);
k = ++i, armazena em *i* o valor 4 (incrementa o valor de *i*) e armazena o valor 4 na variável *j*.

Um exemplo simples: determinando a parte fracionária de um número

Como dissemos na seção 2.3, o armazenamento de um valor de ponto flutuante numa variável do tipo *int* faz com que seja armazenada na variável a parte inteira do valor de ponto flutuante. Isto permite que se extraia facilmente a parte fracionária de um número. Por exemplo, o programa a seguir fornece a parte fracionária de um número dado, calculada como a diferença entre ele e a sua parte inteira.

```
/* Programa que fornece a parte fracionária de um número dado */
#include <stdio.h>
main()
{
float Num, Frac;
int Inteiro;
printf("Digite um numero ");
scanf("%f", &Num);
Inteiro = Num;
Frac = Num - Inteiro;
printf("A parte fracionaria de %f e' %f ", Num, Frac);
}
```

Há que se ter cuidado com números fracionários. Já foi dito que o sistema (e qualquer ambiente para programação) não armazena exatamente todos os números reais, armazenando, de fato, aproximações da maioria deles. Por exemplo, se modificássemos o comando de saída do programa anterior para

```
printf("A parte fracionaria de %f e' %.9f ", Num, Frac);
```

e o executássemos para a entrada 2.41381 teríamos como saída a frase

A parte fracionaria de 2.41381 e' 0.413810015!

O ponto de exclamação (que não faz parte da saída do programa) foi posto pelo fato de que a saída esperada para esta entrada seria 0.41381.

Combinando comandos de atribuição com operadores aritméticos

O comando de atribuição pode ser combinado com operadores aritméticos para substituir atribuições cuja expressão do segundo membro contenha a variável do primeiro membro. Se *x* for o identificador da variável e $ for um operador aritmético, a atribuição

```
x = x $ (expressão);
```

pode ser indicada, simplesmente, por

```
x $= expressão;
```

Por exemplo,

```
x *= 4; equivale a x = x*4;
x += 5; equivale a x = x + 5;
x %= y + 1; equivale a x = x % (y + 1);
x -= 5; equivale a x = x – 5;
x /= 2; equivale a x = x/2;.
```

De acordo com o objetivo do livro, evitaremos a utilização destas opções oferecidas pela linguagem C, por entendermos que elas podem dificultar a legibilidade do comando. No nosso entendimento, só programadores mais experientes devem usar estes recursos.

Lendo caracteres

Alem da possibilidade de se dar entrada em caracteres através da função *scanf()* com código de conversão "%c", pode-se dar entrada em caracteres utilizando-se as funções *getch()* e *getche()* cujos cabeçalhos encontram-se no arquivo *conio.h*. Para a execução destas funções é necessário que se acione uma tecla; quando isto é feito, o caractere correspondente é retornado pela função e pode então ser armazenado numa variável do tipo *char* através de um comando de atribuição. A diferença entre estas funções é que na primeira o caractere digitado não aparece na tela de trabalho, o que acontece com a segunda função. Por

exemplo, a execução do programa

```
#include <stdio.h>
#include <conio.h>
main()
{
char c;
c = getch();
printf("Voce digitou a letra  %c \n", c);
}
```

digitando-se a letra A deixa a tela de trabalho da seguinte forma

```
Voce digitou a letra A
```

enquanto que a execução do programa

```
#include <stdio.h>
#include <conio.h>
main()
{
char c;
c = getche();
printf("Voce digitou a letra  %c \n", c);
}
```

deixa a tela de trabalho da seguinte forma:

```
A
Você digitou a letra A
```

2.10 Exemplos Parte I

1. Voltando ao programa do cálculo da média de três números dados, observe que a média foi calculada e exibida, mas não foi armazenada. Se este programa fizesse parte de um programa maior (e isto normalmente acontece! Não se usa computação para uma questão tão simples!) e esta média fosse necessária em outra parte do programa, aquele trecho teria que ser rescrito. É uma boa prática, portanto, que resultados finais de processamento sejam armazenados em variáveis, sendo então os conteúdos destas variáveis exibidos através da função *printf()*. Assim, o programa referido ficaria melhor escrito da seguinte forma.

```
/* Programa que determina a média de três números dados */
#include <stdio.h>
main()
{
float a, b, c, Media;
puts("Digite três números");
scanf("%f %f %f", &a, &b, &c);
Media = (a + b + c)/3;
printf("A media dos números %f , %f e %f é igual a %f ", a, b, c, Media);
}
```

2. Agora apresentaremos um programa que recebendo um número inteiro como entrada fornece o algarismo da casa das unidades deste número, questão discutida no capítulo 1. Como vimos naquele capítulo, o algarismo procurado é o resto da divisão do número dado por 10. Temos então o seguinte programa (no capítulo 6 veremos um programa que necessita da solução desta questão).

```
/* Programa que determina o algarismo da casa das unidades de um inteiro dado */
#include <stdio.h>
main()
{
```

```
int Num, Unidades;
printf("Digite um inteiro");
scanf("%d", &Num);
Unidades = Num % 10;
printf("O algarismo da casa das unidades de %d e' %d ", Num, Unidades);
}
```

3. Se quiséssemos um programa para inverter um número com dois algarismos (por exemplo, se a entrada fosse 74, a saída deveria ser 47) poderíamos utilizar o seguinte fato: se x e y são os algarismos de um número (casa das dezenas e casa das unidades, respectivamente), então este número é x . 10 + y. Assim, a inversão seria y . 10 + x e bastaria extrair os dois algarismos do número dado e utilizar a expressão acima. A extração do algarismo da casa das unidades foi mostrada no exemplo anterior. E o algarismo da casa das dezenas? Basta ver que ele é o quociente da divisão do número por 10 e este quociente é obtido através do operador / com operandos inteiros. Temos então o seguinte programa.

```
/* Programa que inverte um número com dois algarismos */
#include <stdio.h>
main()
{
int Num, Unidades, Dezenas, Invertido;
printf("Digite um inteiro com dois algarismos");
scanf("%d", &Num);
Unidades = Num % 10;
Dezenas = Num/10;
Invertido = Unidades * 10 + Dezenas;
printf("O invertido de %d e' %d ", Num, Invertido);
}
```

Dificilmente o caro leitor vai escrever um programa com este objetivo (para que serve inverter um número com dois algarismos?). Esta questão e algumas outras estão sendo discutidas aqui apenas como exemplos para o desenvolvimento da lógica de programação e pelo fato de que podem ser trechos de programas maiores, o que será mostrado no próximo exemplo.

4. Imagine que queiramos um programa que determine o menor múltiplo de um inteiro dado maior do que um outro inteiro dado. Por exemplo, se a entrada fosse 13 e 100, a saída deveria ser 104 (104 é o menor múltiplo de 13 que é maior que 100). Como

dividendo = divisor x quociente + resto e resto < divisor,

temos que o valor da expressão

dividendo - resto + divisor

é o múltiplo procurado.

```
/*Programa que determina o menor múltiplo de um inteiro maior que outro inteiro*/
#include <stdio.h>
main()
{
int Num, Divisor, MenMultiplo;
printf("Digite o inteiro do qual o número procurado deve ser  múltiplo");
scanf("%d", &Divisor);
printf("Digite o inteiro que deve ser menor que o múltiplo \n");
scanf("%d", &Num);
MenMultiplo = Num - Num % Divisor + Divisor;
printf("O menor multiplo de %d maior do que %d e' %d \n",  Divisor, Num, MenMultiplo);
}
```

5. Vamos utilizar o raciocínio desenvolvido no exemplo anterior para escrever um programa que será parte fundamental de uma aplicação prática a ser discutida no próximo capítulo. O exemplo mostrará também algo mais ou menos óbvio, mas que deve ser destacado: um programador só é capaz de escrever um programa que resolva um determinado problema se ele souber resolver o tal problema "na mão", ou seja,

com a utilização apenas de lápis e papel.

Trata-se de um programa para determinar a quantidade de múltiplos de um inteiro dado k compreendidos (estritamente) entre dois inteiros x e y dados. Por exemplo, se a entrada for $k = 7$, $x = 10$ e $y = 42$, o programa deve retornar a mensagem *a quantidade de múltiplos de 7 compreendidos entre 10 e 42 é 4* (que são 14, 21, 28, 35).

Uma solução "na mão" desta questão utiliza *progressões aritméticas*, assunto da Matemática que é estudada no Ensino Médio. Uma *progressão aritmética* de *primeiro termo* a_1 e *razão* r é uma *sequência* de números $a_1, a_2, ..., a_n$ cuja diferença entre dois termos consecutivos é constante e igual a r. É fácil provar que $a_n = a_1 + (n - 1)r$.

Na nossa questão, a_1 é o menor múltiplo de k maior que x (exemplo anterior), a_n é o maior múltiplo de k menor que y e r é igual a k. É fácil ver que o maior múltiplo de k menor que y é dado por $(y - 1) - (y - 1)$ % k, sendo $y - 1$ utilizado para gerenciar o caso em que y é múltiplo de k, já que queremos múltiplos de k menor que y.

```
/*Programa que determina o número de múltiplos de um inteiro k situados entre dois inteiros x e y*/
#include <stdio.h>
main()
{
int i, x, y, a, k, NumMultiplos = 0;
printf("Digite os inteiros x e y (y > x)");
scanf("%d %d", &x, &y);
a = y - 1;
printf("Digite o inteiro k \n");
scanf("%d", &k);
NumMultiplos = (a - a % k - x + x % k)/k;
printf("O número de multiplos de  %d compreendidos entre %d e %d e' %d \n",  k, x, y, NumMultiplos);
}
```

6. O programa a seguir, além de ser muito interessante no sentido do desenvolvimento da lógica de programação, será utilizado (a sequência de comandos da função *main()*) em outros programas. Seu objetivo é permutar os conteúdos de duas variáveis. Ou seja, suponhamos que, através de comandos de entrada o programa armazenou nas variáveis x e y os valores 7 e 18 e pretendamos que o programa faça com que o conteúdo de x passe a ser 18 e o de y passe a ser igual a 7. À primeira vista, bastaria a sequência de comandos

```
x = y;
y = x;
```

Ocorre que, quando o segundo comando fosse executado, o primeiro já teria sido e o conteúdo de x não seria mais o original. No nosso exemplo, teríamos a seguinte situação

x	y
17	8
8	
	8

e a permuta não teria sido feita, além do fato de que o conteúdo original de x teria sido perdido. Uma alternativa é considerar uma variável auxiliar que "guarde" o conteúdo de x antes que este seja substituído pelo conteúdo de y. Teríamos assim o seguinte programa.

```
/* Programa que permuta os conteúdos de duas variáveis */
#include <stdio.h>
main()
{
float x, y, Aux;
printf("Digite os dois numeros ");
scanf("%f %f", &x, &y);
printf("Entrada x =  %0.1f, y = %0.1f \n", x, y);
Aux = x;
```

```
x = y;
y = Aux;
printf("Saida x = %0.2f, y = %0.2f \n", x, y);
}
```

Cabe observar que a permuta dos conteúdos pode ser obtida sem a utilização da variável *Aux*. Isto é deixado para que o leitor descubra a solução, sendo apresentado como exercício proposto.

2.11 Funções de biblioteca

Como dissemos na seção 2.5, os compiladores C oferecem diversas funções com objetivos pré-determinados e que podem ser executadas durante a execução de um programa. Para isto a execução da função deve ser solicitada no programa como uma instrução, como operando de uma expressão ou como argumento de outra função (a solicitação da execução de uma função é normalmente chamada de *ativação* ou *chamada* da função). Para que o programador possa colocar no seu programa uma instrução que ative uma função é necessário que ele conheça o *identificador* da função, quantos e de que tipo são os *argumentos* com que elas devem ser ativadas e o tipo de valor que ela *retorna* ao programa quando termina sua execução (como já foi dito, este conjunto constitui o *protótipo* da função). A definição de uma função pré-definida se faz através da seguinte sintaxe.

Identificador da função(Lista de argumentos)

sendo que a lista de argumentos pode ser vazia. A tabela a seguir apresenta algumas das funções pré-definidas dos compiladores C, indicando o tipo dos seus argumentos e comentando o seu valor de retorno.

Tabela 12 Algumas funções de biblioteca

Identificador	Argumentos	O que retorna
fabs(x)	double	Valor absoluto do argumento x
acos(x)	double	Arco cujo valor do co-seno é o argumento x
asin(x)	double	Arco cujo valor do seno é o argumento x
atan(x)	double	Arco cujo valor da tangente é o argumento x
cos(x)	double	Co-seno do argumento x
log(x)	double	Logaritmo natural do argumento x
log10(x)	double	Logaritmo decimal do argumento x
pow(x, y)	double, double	Argumento x elevado ao argumento y
pow10(x)	int	10 elevado ao argumento x
random(x)	int	Um número aleatório entre 0 e x - 1
sin(x)	double	Seno do argumento x
sqrt(x)	double	Raiz quadrada do argumento x
tan(x)	doublé	Tangente do argumento x
tolower(x)	char	Converte o caractere x para minúsculo
toupper(x)	char	Converte o caractere x para maiusculo

O protótipo da função *random()* se encontra no arquivo *stdlib.h* e os protótipos das funções *tolower()* e *toupper()* estão no arquivo *ctype.h*. Os protótipos das outras funções estão no arquivo *math.h* que, como seu nome indica, contém os protótipos das funções matemáticas. Para que a função *random()* seja ativada é necessário que sua ativação seja precedida pela ativação da função *randomize()* que ativa o *gerador de número aleatório*. Por exemplo, o programa abaixo exibirá um número aleatório entre 0 e 99.

```
/* programa que exibe, aleatoriamente, um número entre 0 e 99 */
#include <stdio.h>
#include <stdlib.h>
main()
{
int x;
randomize();
x = random(100);
```

```
printf("%d \n", x);
}
```

O exemplo a seguir, além de pretender motivar o próximo capítulo, ressalta uma observação já feita anteriormente: um programador só é capaz de escrever um programa que resolva um determinado problema se ele souber resolver o tal problema "na mão", ou seja, com a utilização apenas de lápis e papel. Trata-se de um programa que calcule a área de um triângulo, dados os comprimentos dos seus lados. Naturalmente, só é capaz de escrever este programa aquele que conhecer a fórmula abaixo, que dá a área do triângulo cujos lados têm comprimentos *a*, *b* e *c*:

$$S = \sqrt{p.(p - a).(p - b).(p - c)}$$

onde

$$p = \frac{a + b + c}{2}$$

é o *semiperímetro* do triângulo. Com isto, temos o seguinte programa.

```
/*Programa que determina a área de um triângulo de lados de comprimentos dados*/
#include <stdio.h>
#include <math.h>
main()
{
float x, y, z, Area, SemiPer;
printf("Digite os comprimentos dos lados do triangulo");
scanf("%f %f %f", &x, &y, &z);
SemiPer = (x + y + z)/2;
Area = sqrt(SemiPer * (SemiPer - x) * (SemiPer - y) * (SemiPer - z));
printf("A area do triangulo de lados %f , %f e %f e' igual a %f \n", x, y, z, Area);
}
```

Se este programa for executado com entrada 3, 4 e 5 temos SemiPer = 6 e

$$S = \sqrt{6.(6 - 3).(6 - 4).(6 - 5)} = \sqrt{36}$$

e, como era de se esperar, a área do triângulo cujos lados têm comprimento 3, 4 e 5 unidades de comprimento é igual a 6 unidades de área.

Agora, se este programa fosse executado para entrada 1, 2 e 5 teríamos SemiPer = 4, $S = \sqrt{4.(4 - 1).(4 - 2).(4 - 5)} = \sqrt{-24}$ e ocorreria erro de execução pois o sistema (como era de se esperar) não calcula raiz quadrada de número negativo.

O que acontece é que nem sempre três números podem ser comprimentos dos lados de um triângulo (a matemática prova que isto só acontece se cada um deles for menor do que a soma dos outros dois). Assim, o comando que calcula a *Area* só deveria ser executado se os valores digitados para *x*, *y*, e *z* pudessem ser comprimentos dos lados de um triângulo.

2.12 Exercícios propostos

1. Avalie cada uma das expressões abaixo.
a) (-(-9) + sqrt((-9)*(-9) - 4*3*6))/(2*3).
b) ((pow(3, 2) == 9) && (acos(0) == 0)) || (4 % 8 == 3).

2. Escreva programas para
a) Converter uma temperatura dada em graus Fahrenheit para graus Celsius.
b) Gerar o *invertido* de um número com três algarismos (exemplo: o *invertido* de 498 é 894).
c) Somar duas frações ordinárias, fornecendo o resultado em forma de fração.
d) Determinar o maior múltiplo de um inteiro dado menor do que ou igual a um outro inteiro dado

(exemplo: o maior múltiplo de 7 menor que 50 é 49).

e) Determinar o perímetro de um polígono regular inscrito numa circunferência, dados o número de lados do polígono e o raio da circunferência.

3. Escreva um programa que permute o conteúdo de duas variáveis sem utilizar uma variável auxiliar (ver exemplo 5 da seção 2.9).

4. Uma loja vende seus produtos no sistema entrada mais duas prestações, sendo a entrada maior do que ou igual às duas prestações; estas devem ser iguais, inteiras e as maiores possíveis. Por exemplo, se o valor da mercadoria for R$ 270,00, a entrada e as duas prestações são iguais a R$ 90,00; se o valor da mercadoria for R$ 302,75, a entrada é de R$ 102,75 e as duas prestações são a iguais a R$ 100,00. Escreva um programa que receba o valor da mercadoria e forneça o valor da entrada e das duas prestações, de acordo com as regras acima. Observe que uma justificativa para a adoção desta regra é que ela facilita a confecção e o consequente pagamento dos boletos das duas prestações.

5. Um intervalo de tempo pode ser dado em dias, horas, minutos, segundos ou sequências "decrescentes" destas unidades (em dias e horas; em horas e minutos; em horas, minutos e segundos), de acordo com o interesse de quem o está manipulando. Escreva um programa que converta um intervalo de tempo dado em segundos para horas, minutos e segundos. Por exemplo, se o tempo dado for 3 850 segundos, o programa deve fornecer 1 h 4 min 10 s.

6. Escreva um programa que converta um intervalo de tempo dado em minutos para horas, minutos e segundos. Por exemplo, se o tempo dado for 145.87 min, o programa deve fornecer 2 h 25 min 52.2 s (vale lembrar que o ponto é o separador da parte inteira).

7. Um programa para gerenciar os saques de um caixa eletrônico deve possuir algum mecanismo para decidir o número de notas de cada valor que deve ser disponibilizado para o cliente que realizou o saque. Um possível critério seria o da "distribuição ótima" no sentido de que as notas de menor valor disponíveis fossem distribuídas em número mínimo possível. Por exemplo, se a máquina só dispõe de notas de R$ 50, de R$ 10, de R$ 5 e de R4 1, para uma quantia solicitada de R$ 87, o programa deveria indicar uma nota de R$ 50, três notas de R$ 10, uma nota de R$ 5 e duas notas de R$ 1. Escreva um programa que receba o valor da quantia solicitada e retorne a distribuição das notas de acordo com o critério da distribuição ótima.

8. De acordo com a Matemática Financeira, o cálculo das prestações para amortização de um financiamento de valor F em *n* prestações e a uma taxa de juros *i* é dada pela fórmula $P = F/a_{n\neg i}$, onde $a_{n\neg i} = ((1 + i)^n - 1)/(i \,.\, (1 + i)^n)$. Escreva um programa que determine o valor das prestações para amortização de um financiamento, dados o valor do financiamento, o número de prestações para amortização e a taxa de juros.

Observação

Propostas de soluções dos exercícios propostos podem ser solicitadas através de mensagem eletrônica para jaime@ccen.ufal.br com assunto RESPOSTAS LIVRO C, anexando o formulário abaixo devidamente preenchido.

Nome	Categoria[1]	Instituição[2]	Curso[2]	Cidade/Estado

[1]Categoria: docente, estudante, autodidata

[2]Se docente ou estudante

3 Estruturas de seleção

3.1 O que é uma estrutura de seleção

O último exemplo do capítulo anterior apresentava um programa para calcular a área de um triângulo, dados os comprimentos dos seus lados. Foi visto que o comando que calculava a área solicitada só devia ser executado com a certeza anterior de que os valores dados como entrada poderiam ser comprimentos dos lados de um triângulo. Ou seja, o tal comando só deveria ser executado *se* $x < y + z$ *e* $y < x + z$ *e* $z < x + y$, condição que garante que os valores armazenados nas variáveis x, y e z são comprimentos dos lados de um triângulo. Assim, em algumas situações, alguns comandos só devem ser executados se alguma condição for satisfeita.

É muito fácil encontrar situações em que a execução de uma ou mais instruções deve estar condicionada ao fato de que uma condição seja satisfeita. Por exemplo, veremos algoritmos para ordenar uma relação de números que necessitam colocar em ordem crescente os conteúdos de duas variáveis. É óbvio que para ordenar em ordem crescente os conteúdos de duas variáveis x e y só é necessário se fazer alguma coisa se o conteúdo de x for maior que o conteúdo de y, ou seja, se $x > y$.

Há situações também em que há necessidade de que se faça uma escolha entre duas ou mais sequências de instruções qual a sequência deve ser executada. Por exemplo, se pretendemos verificar se um número n é par podemos determinar o resto da divisão de n por 2. Se este resto for zero, então o número é par. Se este resto for 1, o número é ímpar.

Vale lembrar que os algoritmos que o viver exige que executemos diuturnamente são pontuados de escolhas e decisões: *se não chover, iremos para a praia*, *se chover, iremos para o shopping*; *se estiver fazendo frio, vista um casaco*.

A verificação de que uma condição é satisfeita e, a partir daí, uma determinada sequência de comandos deve ser executada é chamada de *estrutura de seleção*, *estrutura de decisão* ou *comando de seleção*.

3.2 O comando *if*

O comando *if* é uma estrutura de decisão que decide se uma sequência de comandos será ou não executada. Sua sintaxe é

```
if (Expressão)
{
sequência de comandos
}
```

sendo os delimitadores opcionais se a sequência de comandos contém um único comando.

A semântica deste comando é muito simples: se o valor da *Expressão* for diferente de zero, o sistema executará a sequência de comandos; caso contrário o sistema não executará a sequência de comandos e a instrução após o comando *if* passa a ser executada.

Por exemplo, se queremos um programa que determine o maior de dois números dados, podemos supor que o primeiro deles é o maior, armazenando-o numa variável *Maior* e depois, através de um comando *if*, verificar se o maior procurado é o segundo dos números dados; neste caso o conteúdo da variável *Maior* deve ser alterado.

```
/*Programa para determinar o maior de dois números dados */
#include <stdio.h>
main()
{
float a, b, Maior;
printf("Digite os dois numeros");
scanf("%f %f", &a, &b);
Maior = a;
```

```
if (b > a)
   Maior =  b;
printf("O maior dos numeros %f , %f  e'  %f ", a, b, Maior);
}
```

Um outro exemplo de utilização do comando *if* aparece num programa que pretenda ordenar os conteúdos de variáveis *x* e *y*. Para isto só há de necessidade de se realizar alguma ação se o conteúdo de *y* for maior do que o conteúdo de *x*. Neste caso, o que deve ser feito é a permuta dos conteúdos de *x* e de *y*. Temos então o seguinte programa.

```
/* Programa para ordenar os conteúdos de duas variáveis */
#include <stdio.h>
main()
{
float x, y, Aux;
printf("Digite os dois numeros");
scanf("%f %f", &x, &y);
printf("Conteudos originais de x e de y: %f , %f \n: ", x, y);
if (y < x)
   {
   Aux = x;
   x = y;
   y = Aux;
   }
printf("Conteudos de x e de y ordenados: %f , %f: \n", x, y);
}
```

Observe que a sequência de comandos

```
Aux =x;
x = y;
y = Aux;
```

realiza a permuta dos conteúdos das variáveis *x* e *y*, como discutido em exemplo do capítulo anterior.

3.3 O comando *if else*

O comando *if else* é uma estrutura de decisão que decide entre duas sequências de comandos qual vai ser executada, sendo definido através da seguinte sintaxe:

```
if (Expressão)
   {
   Sequência de comandos 1
   }
else
   {
   Sequência de comandos 2
   }
```

A semântica deste comando é a seguinte: se o valor de *Expressão* for diferente de zero, o sistema executará a sequência de comandos 1; caso contrário o sistema executará a sequência de comandos 2.

Por exemplo, se queremos um programa que verifique a paridade de um número dado, poderíamos verificar se o resto da divisão do número por dois é igual a 0. Se isto for verdade, o número é par; se não for verdade, o número dado é ímpar.

```
/* Programa para verificar se um número e' par */
#include <stdio.h>
main()
{
```

```
int x, y;
printf("Digite o numero");
scanf("%d", &x);
if (x % 2 == 0)
  printf("%d e' par \n", x);
else
  printf("%d e' impar \n", x);
}
```

Mesmo considerando que os compiladores da linguagem C não consideram espaços nem mudanças de linha, observe que estamos procurando escrever cada instrução em uma linha e a sequência vinculada à estrutura de decisão com uma tabulação diferente da tabulação em que estão postos o *if* e o *else*. Esta forma de se editar um programa, chamada *indentação*, deve ser praticada por todo programador pois ela facilita sobremaneira a legibilidade dos programas. Se o programa acima fosse digitado da forma seguinte

```
/* Programa para verificar se um número é par*/
#include <stdio.h>
main(){
int x, y; printf("Digite o numero"); scanf("%d", &x);
if (x % 2 == 0) printf("%d e' par \n", x); else
printf("%d e' impar \n", x);
}
```

ele seria executado da mesma forma, porém a sua legibilidade estaria prejudicada.

3.4 O operador condicional ternário

Quando as duas opções de um comando *if else* contêm apenas uma atribuição a uma mesma variável, pode-se utilizar o *operador condicional ternário* que possui a seguinte sintaxe:

Variável = Expressão lógica ? Expressão 1 : Expressão 2;

Na execução deste comando a *Expressão lógica* é avaliada e se for diferente de zero o valor da *Expressão 1* é atribuído à *Variável*; caso contrário, o valor da *Expressão 2* é atribuído. Por exemplo, se x, y e *Maior* são três variáveis do tipo *float* o armazenamento do maior dos conteúdos de x e de y na variável *Maior* poderia ser obtido com a seguinte atribuição:

```
Maior = (x > y) ? x : y;
```

Como um outro exemplo, para se armazenar na variável *AbsNum* o valor absoluto do conteúdo de uma variável *Num* (sem utilizar a função *fabs()*) bastaria o comando:

```
AbsNum = (Num >= 0) ? Num : -Num;
```

3.5 Exemplos Parte II

0. De um modo geral, as ligações telefônicas são cobradas pelas suas durações. O sistema registra os instantes em que a ligação foi iniciada e concluída e é acionado um programa que determina o intervalo de tempo decorrido entre aqueles dois instantes dados. O programa abaixo recebe dois instantes dados em horas e minutos e determina o intervalo de tempo (em horas e minutos) decorrido entre eles.

```
/*Programa que determina o intervalo de tempo decorrido entre dois instantes*/
include <stdio.h>
main()
{
int h1, min1, h2, min2, h, min;
puts("Digite o instante inicial (horas e minutos)");
scanf("%d %d", &h1, &min1);
```

```
puts("Digite o instante final");
scanf("%d %d", &h2, &min2);
h = h2 - h1;
min = min2 - min1;
if ((h < 0) || ((h == 0) && (min < 0)))
        puts("\aDados invalidos! O segundo instante é anterior ao primeiro");
else
        {
        if (min < 0)
                {
                h = h - 1;
                min = min + 60;
                }
        printf( "Entre os instantes %dh %dmin e %dh  %dmin passaram-se %dh %dmin", h1, min1, h2,
min2, h,  min);
        }
}
```

1. No último exemplo do capítulo 2, apresentamos um programa que calculava a área de um triângulo, dados os comprimentos dos seus lados. No final dele, mostramos que o mesmo não fornecia respostas satisfatórias para todas as entradas e comentamos que o cálculo da área deveria ser precedido da verificação de que os dados de entrada são de fato comprimentos dos lados de um triângulo. O programa referido, escrito agora de forma completa e correta, seria o seguinte.

```
/* Programa para calcular a área de um triângulo*/
#include <stdio.h>
#include <math.h>
main()
{
float x, y, z, Area, SemiP;
printf("Digite os comprimentos dos lados do triangulo");
scanf("%f %f %f", &x, &y, &z);
if ((x < y + z) && (y < x + z) && (z < x + y))
        {
        SemiP = (x + y + z)/2;
        Area = sqrt(SemiP * (SemiP - x) * (SemiP - y) * (SemiP - z));
        printf("A area do triangulo de lados %f , %f e %f e' igual a %f \n", x, y, z, Area);
        }
else
        printf("Os números %f, %f %f não podem ser comprimentos dos lados de um triângulo\n", x, y,
z);
}
```

2. Programas que manipulam datas (por exemplo, um programa que determine o número de dias entre duas datas dadas) contêm trechos que verificam se um ano dado é bissexto. Sabendo que um ano é bissexto se ele é múltiplo de quatro, teríamos o seguinte programa.

```
/*Programa que verifica se um dado ano é bissexto */
#include <stdio.h>
main()
{
int Ano;
printf("Digite o ano");
scanf("%d", &Ano);
if (Ano % 4 == 0)
        printf("%d e' bissexto %d \n", Ano);
else
        printf("%d não e' bissexto %d \n", Ano);
}
```

Rigorosamente falando, há anos múltiplos de quatro que não são bissextos. São aqueles múltiplos de 100 que não são múltiplos de 400. Por exemplo, o ano 2000 foi um ano bissexto, mas o ano de 2100 não será. Para que o programa detecte estas exceções, a expressão lógica que controla o comando *if* deve ser ampliada e talvez seja mais fácil considerar a condição para que um ano não seja bissexto: não deve ser múltiplo de quatro ou se for múltiplo de 100 não deve ser múltiplo de 400. Observe que agora optamos por uma expressão lógica que garantisse o fato de que o ano dado não é bissexto.

```
/* Programa que verifica se um dado ano é bissexto */
#include <stdio.h>
main()
{
int Ano;
printf("Digite o ano");
scanf("%d", &Ano);
if ((Ano % 4 != 0) || ((Ano % 100 == 0) && (Ano % 400 != 0)))
        printf("%d nao e' bissexto  \n", Ano);
else
        printf("%d e' bissexto  \n", Ano);
}
```

3. O programa para ordenar os conteúdos de duas variáveis, visto na seção 3.2, é um caso muito particular da questão mais geral da *ordenação* de uma relação de números ou de nomes, problema que tem vasta aplicação na vida prática, principalmente na ordenação de uma lista de nomes (este problema também é conhecido como *classificação*). Para a solução geral existem diversos algoritmos com este objetivo. No capítulo 7 teremos oportunidade de discutir programas baseados em alguns destes algoritmos. Por enquanto, vejamos um programa que ordene três números dados. Além de exemplificar o comando *if*, o programa abaixo mostra como se pode (e se deve) utilizar raciocínios anteriores para se escrever programas.

Seja então um programa que receba três números inteiros, armazene-os nas variáveis *x*, *y* e *z* e que ao final da sua execução deixe os conteúdos de *x*, de *y* e de *z* na ordem crescente. Uma ideia bem interessante é armazenar na variável *x* o menor dos números e em seguida ordenar os conteúdos de *y* e de *z*, que é exatamente o problema de ordenar os conteúdos de duas variáveis, que foi referido acima. Obviamente, para se executar a primeira ação pretendida (armazenar na variável *x* o menor dos números) só é necessário se fazer alguma coisa se o valor de *x* já não for o menor dos números dados, ou seja, se $x > y$ ou $x > z$. Nesta hipótese, o menor deles é *y* ou *z* e este menor deve ser permutado com *x*. Temos então o seguinte programa.

```
/* Programa para ordenar três números dados*/
#include <stdio.h>
main()
{
float x, y, z, Aux;
printf("Digite os tres numeros");
scanf("%f %f %f", &x, &y, &z);
printf("Numeros dados: %f , %f , %f \n", x, y, z);
if ((x > y) || (x > z))         /* verifica se x não é o menor */
        if (y < z)                              /* neste caso y é o menor */
                {
                Aux = x;                        /* troca os conteúdos de x e de y */
                x = y;
                y = Aux;
                }
        else                                    /* neste caso z é o menor */
                {
                Aux = x;                        /* troca os conteúdos de x e de z */
                x = z;
                z = Aux;
                }
if (y > z)                                      /* verifica se z e y ainda não estão ordenados */
        {
        Aux = y;                                /* troca o conteúdo de y e de z */
```

```
	y = z;
	z = Aux;
	}
printf("Numeros ordenados: %f , %f , %f \n", x, y, z);
}
```

Observe que se a expressão lógica do primeiro comando *if* for verdadeira, o sistema executará outro comando *if*. Neste caso, dizemos que os comandos estão *aninhados*. Observe também que escrevemos no programa algumas frases explicativas das ações pretendidas. Esta frases são chamadas *comentários* e devem ser escritas entre os pares de caracteres /* e */. Quando o compilador encontra o par de caracteres /* procura um outro par */ e desconsidera tudo o que vem entre os dois pares. Isto permite que o programador deixe registrado no próprio programa as observações que ele achar conveniente. Como a edição dos programas com indentação, a prática de se colocar comentários nos programas é muito importante. Como os programas discutidos neste livro serão precedidos de explicações prévias, a utilização de comentários aqui vai se restringir à indicação do objetivo do programa (como já víamos fazendo).

A ação realizada pela primeira estrutura de decisão do programa acima pode ser obtida através de outro algoritmo. A ideia é a seguinte: coloca-se na variável *x* o menor dos valores inicialmente armazenados nas variáveis *x* e *y*. Em seguida, repete-se o raciocínio com os valores armazenados (agora) em *x* e em *z*.

```
/* Programa para ordenar três números dados (versão 2)*/
#include <stdio.h>
main()
{
float x, y, z, Aux;
printf("Digite os tres numeros");
scanf("%f %f %f", &x, &y, &z);
printf("Numeros dados: %f , %f , %f \n", x, y, z);
if (x > y)
	{
	Aux = x;
	x = y;
	y = Aux;
	}
if (x > z)
	{
	Aux = x;
	x = z;
	z = Aux;
	}
if (y > z)
	{
	Aux = y;
	y = z;
	z = Aux;
	}
printf("Numeros ordenados: %f , %f , %f \n", x, y, z);
}
```

4. Um outro exemplo que ilustra muito bem a utilização do comando *if* é um programa para determinar as raízes de uma equação do segundo grau. Sabemos da matemática que uma equação $ax^2 + bx + c = 0$ só tem raízes reais se $b^2 - 4ac < 0$. Assim, um programa para encontrar as raízes reais (deixaremos o caso completo da determinação das raízes reais e complexas como exercício proposto) poderia ser o seguinte.

```
/*Programa que calcula as raízes de uma equação do segundo grau */
#include <stdio.h>
#include <math.h>
main()
{
float a, b, c, Delta, x1, x2;
```

```
printf("Digite os coeficientes");
scanf("%f %f %f", &a, &b, &c);
if (a != 0)
        {
        Delta = b*b - 4*a*c;
        if (Delta >= 0)
            {
            x1 = (-b + sqrt(Delta))/(2*a);
            x2 = (-b - sqrt(Delta))/(2*a);
            printf("As raizes da equacao de coeficientes %f , %f e %f sao %f e %f ", a, b, c, x1, x2);
            }
  else
            printf("A equacao nao tem raizes reais");
  }
else
  printf("A equacao nao e do segundo grau");
}
```

5. Imaginemos agora uma escola que adote no seu processo de avaliação a realização de duas avaliações bimestrais e que o regime de aprovação dos alunos seja o seguinte:

i) Se a média das avaliações bimestrais for superior ou igual a 7,0, o aluno está *aprovado*, com média final igual à média das avaliações bimestrais.

ii) Se a média das avaliações bimestrais for inferior a 5,0, o aluno está *reprovado*, com média final igual à média das avaliações bimestrais.

iii) Não ocorrendo nenhum dos casos anteriores, o aluno se submete a uma *prova final* e a sua média final será a média ponderada desta prova final (com peso 4) e a média das avaliações bimestrais (com peso 6). Neste caso, o aluno estará aprovado se a sua média final for superior ou igual a 5,5.

O programa abaixo recebendo as notas das avaliações bimestrais e, se for o caso, a nota da prova final, fornece a média final do aluno e a sua condição em relação à aprovação.

```
/* Programa para verificar aprovação de um aluno*/
#include <stdio.h>
main()
{
float Bim1, Bim2, MedBim, PrFinal, MedFinal;
printf("Digite as duas notas bimestrais");
scanf("%f %f ", &Bim1, &Bim2);
MedBim = (Bim1 + Bim2)/4;
MedFinal = MedBim;
if ((MedBim < 7) && (MedBim >= 5))
  {
  printf("Digite a nota da prova final");
  scanf("%f", &PrFinal);
  MedFinal = (MedBim * 6 + PrFinal * 4)/10;
  }
if (MedFinal > 5.5)
  printf("Aluno aprovado com media final %.2f \n", MedFinal);
else
  printf("Aluno reprovado com media final %0.2f \n", MedFinal);
}
```

6. Para um exemplo de um programa que utiliza vários comandos *if* aninhados, suponhamos que uma empresa decidiu dar um aumento escalonado a seus funcionários de acordo com a seguinte regra: 13% para os salários inferiores ou iguais a R$ 200,00; 11% para os salários situados entre R$ 200,0 e R$ 400,00 (inclusive); 9 % para os salários entre R$ 400,00 e R$ 800,00 (inclusive) e 7% para os demais salários. Um programa que receba o salário atual de um funcionário e forneça o valor do seu novo salário poderia ser o seguinte.

```
/*Programa para atualizar salários*/
```

```
#include <stdio.h>
main()
{
float SAtual, SNovo, Indice;
printf("Digite o salário atual");
scanf("%f", &SAtual);
if (SAtual <= 200)
  Indice = 1.13;
else
  if (SAtual <= 400)
      Indice = 1.11;
  else
      if (SAtual <= 800)
          Indice = 1.09;
      else
          Indice = 1.07;
SNovo = SAtual*Indice;
printf("Atual =  %.2f \n Novo  = %.2f \n" , SAtual, SNovo);
}
```

Observe que a sequência associada à opção *else* é iniciada com um outro comando *if*. Alguns autores preferem destacar um fato como este definindo um "novo comando" denominando-o *else if*.

7. Um outro exemplo que utiliza comandos de seleção aninhados e em que a escolha da expressão lógica que controlará o comando *if* é importante é um programa que determine o número de dias de um mês (um programa como este seria parte integrante de um programa que manipulasse datas). Como os meses de trinta dias são quatro e os de trinta e um dias são sete, usamos os primeiros para o controle do comando de seleção.

```
/* Programa que determina o número de dias de um mês dado */
#include <stdio.h>
main()
{
int Mes, Ano, NumDias;
printf("Digite o mes");
scanf("%d", &Mes);
if ((Mes == 4 ) || (Mes == 6) || (Mes == 9) || (Mes == 11))
  NumDias = 30;
else
  if (Mes == 2)
      {
      printf("Digite o ano");
      canf("%d", &Ano);
      if (Ano % 4 != 0)
          NumDias = 28;
      else
          NumDias = 29;
      }
  else
      NumDias = 31;
printf("O mes %d tem %d dias", Mes, NumDias);
}
```

No capítulo 6 veremos que o programa acima pode ser bastante simplificado.

3.6 O comando *switch*

Muitos programas são desenvolvidos de modo que eles podem realizar várias tarefas, de forma independente. Por exemplo, um programa que gerencie um caixa eletrônico de um banco deve oferecer ao usuário algumas opções em relação à ação que ele pretende realizar na sua conta como a emissão do saldo atual, a emissão de um extrato, a realização de um saque e a realização de um depósito. É comum que um programa que permita a realização de várias tarefas inicie apresentando ao usuário um *menu de opções* com a indicação das diversas tarefas que o programa pode executar e a permissão de que o usuário escolha a tarefa pretendida. Como, em geral, são várias as opções disponíveis (cada uma delas com uma sequência específica de comandos) e só uma das opções será a escolhida, é necessária uma estrutura que decide entre várias sequências de comandos qual vai ser executada ou quais vão ser executadas.

O comando *switch* tem este objetivo e deve ser escrito com a seguinte sintaxe:

```
switch(Expressão)
{
case constante1 :
  Sequência de instruções 1
case constante2 :
  Sequência de instruções 2
. . .
case constante n :
  Sequência de instruções  n
default :
  Sequência de comando x
}
```

Aí, a *Expressão* argumento do comando deve resultar num valor do tipo *int* ou num valor do tipo *char* e, opcionalmente, a ultima instrução de cada uma das sequências *Sequência de instruções i* é *break*. A semântica deste comando é bem simples: a *Expressão* é avaliada e as sequências de instruções situadas entre o valor da expressão apresentado nos *cases* e um comando *break* ou o delimitador do comando são executadas. Se o valor da *Expressão* for diferente de todas as opções dadas pelas constantes associadas aos *cases*, a sequência de instruções vinculada ao *default* será executada. Por exemplo, o programa

```
#include <stdio.h>
main()
{
int x;
printf("Digite um número inteiro entre 1 e 5 \n");
scanf("%d", &x);
switch (x)
  {
  case 1 : printf("Valor de x: %d  \n", x);
  case 2 : printf("Valor  do dobro de %d: %d  \n", x, 2*x);
  case 3 : printf("Valor do triplo de %d: %d \n", x, 3*x);
  case 4 : printf("Valor do quadruplo de %d: %d \n", x, 4*x);
  default : printf("Valor digitado: %d \n", x);
  }
}
```

executado para x = 1 executa todas as sequências vinculadas aos *cases* fornecendo a seguinte saída:

```
Valor de x: 1
Valor do dobro de 1: 2
Valor do triplo de 1: 3
Valor do quadruplo de 1: 4
Valor digitado: 1
```

Se for executado para x = 3, só as sequências a partir do *case 3* serão executadas e a saída será:

```
Valor do triplo de 3: 9
```

Valor do quadruplo de 3: 12
Valor digitado: 3

e se for executado x = 10 apenas a sequência vinculada à condição *default será a executada* e a saída será:

Valor digitado : 10

Três observações:

1. A sequência de instruções vinculada a uma opção *case* pode ser vazia, caso em que, evidentemente, nada é executado;
2. Se apenas uma sequência de comandos deve ser executada, deve-se encerrá-la com um *break*;
3. A opção *default* é opcional: se ela não aparece no comando e o valor da *Expressão* for diferente de todos os valores disponíveis, nada é executado e a instrução logo após o comando *switch* passa a ser executada.

3.7 Exemplos Parte III

1. O programa para determinar o número de dias de um mês (exemplo 7 da seção anterior) poderia utilizar o comando *switch*:

```
/* Programa para determinar o numero de dias de um mes*/
#include <stdio.h>
main()
{
int Mes, Ano, NumDias;
printf("Digite o mes \n");
scanf("%d", &Mes);
switch (Mes)
  {
  case 2 :
            printf("Digite o ano");
            scanf("%d", &Ano);
            if (Ano % 4 != 0)
                      NumDias = 28;
            else
                      NumDias = 29;
            break;
  case 4 :
  case 6 :
  case 9 :
  case 11 : NumDias = 30; break;
  default : NumDias = 31;
  }
printf("O mes de numero %d tem %d dias \n", Mes, NumDias);
}
```

Observe que se o mês de entrada for 2, o programa pede o ano para determinar se ele é bissexto. Aí, determina o número de dias e a instrução *break* encerra o comando *switch*. Se a entrada for 4, com a sequência de comandos vinculada ao *case* 4 é vazia (e, portanto, não contém *break*) as sequências vinculadas aos *cases* seguintes são executadas até o *break* do *case* 11 (para os meses 4, 6, 9 e 11 o número de dias é igual a 30!). Se a entrada não for 2, 4, 6, 9 e 11 a opção *default* será executada e, portanto, o mês terá 31 dias. Evidentemente, fica faltando discutir a possibilidade de uma entrada inválida como, por exemplo, 13. Isto será discutido no próximo capítulo.

2. Vejamos um exemplo onde a expressão do comando *switch* retorna um valor do tipo *char*. Trata-se da geração de uma calculadora para as quatro operações aritméticas básicas.

```
/*Calculadora eletrônica*/
#include <stdio.h>
```

```
#include <conio.h>
main()
{
float Op1, Op2, Res;
char Operador;
clrscr();
printf("Digite a opera‡ao desejada\n");
scanf("%f %c %f", &Op1, &Operador, &Op2);
switch (Operador)
  {
  case '+':
            Res = Op1 + Op2; break;
  case '-':
            Res = Op1 - Op2; break;
  case '*':
            Res = Op1 * Op2; break;
  case '/':
            if (Op2 != 0)
                      Res = Op1 / Op2; break;
  }
clrscr();
if (Operador == '/' && Op2 == 0)
  printf("Divisao por zero!!!");
else
  printf("%.2f %c %.2f = %.2f \n",  Op1, Operador, Op2, Res);
getch();
}
```

3. Um outro exemplo interessante de utilização do comando *switch* é um programa que determine o dia da semana de uma data dada. Tomando como base o ano de 1600 (em 1582 o Papa Gregorio III instituiu mudanças no calendário então vigente) e sabendo que o dia primeiro daquele ano foi um sábado, para se determinar o dia da semana de uma data dada basta se calcular o número de dias decorridos entre a data dada e o dia 01/01/1600. Como a associação do dia da semana a uma data é periódica, de período 7, o resto da divisão do número de dias referido acima por 7 indica a relação entre o dia da semana procurado e o sábado: se o tal resto for 0 (zero), o dia da semana é sábado; se o resto for 1 o dia da semana é domingo, e assim sucessivamente.

Para calcular o número de dias entre uma data dada e 01/01/1600 basta multiplicar o número de anos por 365 e acrescentar a quantidade de anos bissextos e o número de dias decorridos no ano corrente.

Para calcular a quantidade de anos bissextos entre 1600 e o ano da data dada basta calcular a expressão *Quantidade de múltiplos de 4 – Quantidade de múltiplos de 100* + *Quantidade de múltiplos de 400*, onde *Quantidade de múltiplos de x* refere-se à quantidade de múltiplos de *x* compreendidos entre 1600 e o ano da data dada, como discutido no exemplo 5 da seção 2.10.

Para calcular o número de dias decorridos no ano da data dada basta ...(isto está explicado nos comentários do programa).

```
/* Programa para determinar o dia da semana de uma data dada */
#include <stdio.h>
#include <conio.h>
main()
{
int Dia, Mes, Ano, DiasDoAno, Dias31, AnosBiss, Aux, Mult4, Mult100, Mult400;
long  int Anos, NumDias;
clrscr();
printf("Digite a data no formato dd/mm/aaaa\n");
scanf("%d/%d/%d", &Dia, &Mes, &Ano);
Anos = Ano - 1600;
/* Numero de meses com 31 dias ate o mês dado */
if (Mes < 9)
```

```
  Dias31 = Mes/2;
else
  Dias31 = (Mes + 1)/2;
/*Numero de dias do ano dado, considerando fevereiro com tendo 30 dias*/
DiasDoAno = 30*(Mes - 1) + Dia + Dias31;
/*Retifica o numero de dias de fevereiro*/
if (Mes > 2)
  if ((Ano % 4 != 0) || ((Ano % 100 == 0) && (Ano % 400 != 0)))
          DiasDoAno = DiasDoAno - 2;
  else
          DiasDoAno = DiasDoAno - 1;
/*Numero de anos bissextos entre o ano dado e 1600*/
Aux = Ano - 1;
Mult4 = (Aux - (Aux % 4) - 1600)/4;
Mult100 = (Aux - (Aux % 100) - 1600)/100;
Mult400 = (Aux - (Aux % 400) - 1600)/400;
AnosBiss = Mult4 - Mult100 + Mult400;
/*Numero de dias entre a data dada e 01/01/1600*/
NumDias = Anos*365 + DiasDoAno + AnosBiss;
/*Dia da semana*/
printf("\nData: %d/%d/%d          Dia da semana:", Dia, Mes, Ano);
switch(NumDias % 7)
  {
  case 0 : printf(" Sabado"); break;
  case 1 : printf(" Domingo"); break;
  case 2 : printf(" Segunda"); break;
  case 3 : printf(" Terca"); break;
  case 4 : printf(" Quarta"); break;
  case 5 : printf(" Quinta"); break;
  case 6 : printf(" Sexta"); break;
  }
getch();
}
```

Vale observar que este programa dará uma "resposta" mesmo que a data dada não seja uma data válida, como 29/02/2009 por exemplo. Isto será discutido no próximo capítulo.

Vale observar também que o programa realiza pelo menos duas ações com objetivos específicos e raciocínios próprios: o cálculo do número de anos bissextos entre 1600 e ano da data dada e a determinação do número de dias decorridos no referido ano. No capítulo 5 vamos mostrar que se pode (se deve) escrever *subprogramas* (*funções*) para realizar cada uma destas ações.

3.8 Exercícios propostos

1. Reescreva o programa do exemplo zero da seção 3.5 de modo que os instantes sejam dados (e o intervalo de tempo fornecido) em horas minutos e segundos**.**

2. Escreva um programa que realize arredondamentos de números utilizando a regra usual da matemática: se a parte fracionária for maior do que ou igual a 0,5, o número é arredondado para o inteiro imediatamente superior, caso contrário, é arredondado para o inteiro imediatamente inferior.

3. Escreva um programa para verificar se um inteiro dado é um quadrado perfeito, exibindo, nos casos afirmativos, sua raiz quadrada.

4. Escreva um programa para determinar o maior de três números dados.

5. Escreva um programa para classificar um triângulo de lados de comprimentos dados em *escaleno* (os três lados de comprimentos diferentes), *isósceles* (dois lados de comprimentos iguais) ou *equilátero* (os três lados de comprimentos iguais).

6. Escreva um programa para verificar se um triângulo de lados de comprimentos dados é *retângulo*,

exibindo, nos casos afirmativos, sua *hipotenusa* e seus *catetos*.

7. Escreva um programa para determinar as raízes reais ou complexas de uma equação do segundo grau, dados os seus coeficientes.

8. Escreva um programa para determinar a idade de uma pessoa, em anos meses e dias, dadas a data (dia, mês e ano) do seu nascimento e a data (dia, mês e ano) atual.

9. Escreva um programa que, recebendo as duas notas bimestrais de um aluno da escola referida no exemplo 5 da seção 3.5, forneça a nota mínima que ele deve obter na prova final para que ele seja aprovado.

Observação

Propostas de soluções dos exercícios propostos podem ser solicitadas através de mensagem eletrônica para jaime@ccen.ufal.br com assunto RESPOSTAS LIVRO C, anexando o formulário abaixo devidamente preenchido.

Nome	Categoria[1]	Instituição[2]	Curso[2]	Cidade/Estado

[1]Categoria: docente, estudante, autodidata

[2]Se docente ou estudante

4. Estruturas de repetição

4.1 Para que servem as estruturas de repetição

Um locutor brasileiro ao narrar um jogo de futebol americano nos Estados Unidos recebe a informação do placar eletrônico sobre a temperatura do estádio medida em graus Fahrenheit. Naturalmente, ele deve fornecer aos telespectadores brasileiros a temperatura em graus Celsius. Para isto, o locutor, de posse de um computador, poderia utilizar o programa abaixo, que foi solicitado no primeiro item do segundo exercício da seção 2.12.

```
/*Programa que converte uma temperatura dada em graus Fahrenheit para graus Celsius*/
#include <stdio.h>
main()
{
float Fahrenheit, Celsius;
printf("Digite a temperatura em Fahrenheit");
scanf("%f", &Fahrenheit);
Celsius = 5 * (Fahrenheit - 32)/9;
printf("A temperatura de %.2f  Fahrenheit corresponde a %.2f Celsius ", Fahrenheit, Celsius);
}
```

Se o placar eletrônico indicasse uma temperatura de 60º F, o narrador executaria o programa com a entrada 60 e receberia a saída

A temperatura de 60 graus Fahrenheit corresponde a 15.55 graus Celsius

Certamente, seria mais prático a produção da transmissão do evento disponibilizar para o locutor uma tabela contendo as temperaturas possíveis em graus Fahrenheit e as correspondentes em graus Celsius. A confecção desta tabela poderia ser feita através de um programa que contivesse vários comandos que calculassem para cada temperatura em graus Fahrenheit pretendida a correspondente temperatura em graus Celsius e exibissem estas temperaturas. Neste caso, não haveria necessidade de comando de entrada; porém, para cada temperatura em graus Fahrenheit pretendida, haveria, pelo menos, um comando de atribuição e a chamada da função *printf()*. Se a faixa de temperatura em graus Fahrenheit a ser coberta pela tabela fosse de vinte a oitenta graus, teríamos um programa como o programa abaixo.

```
/*Programa (muito ruim) que gera uma tabela de conversão de temperaturas em graus Fahrenheit para graus Celsius */
#include <stdio.h>
main()
{
int Fahrenheit;
printf("Tabela de conversao graus Fahrenheit/graus Celsius \n");
printf("-------------------------------------------------\n");
printf("\t Fahrenheit \t | \t Celsius\n");
printf("-------------------------------------------------\n");
Fahrenheit = 10;
printf("\t %f \t | \t %f \n", Fahrenheit, 5.0*(Fahrenheit - 32)/9);
Fahrenheit = 11;
printf("\t %f \t | \t %f \n", Fahrenheit, 5.0*(Fahrenheit - 32)/9);
. . .
/*Mais "uma porção" de comandos! */
Fahrenheit = 80;
printf("\t %f \t | \t %f \n", Fahrenheit, 5.0*(Fahrenheit - 32)/9);
}
```

Isto seria contornado se pudéssemos repetir a execução dos comandos que gerariam as temperaturas em graus Fahrenheit e as correspondentes em graus Celsius. A linguagem C possui os comandos *for*; *while* e

do while, chamados *estruturas de repetição* ou *laços*, cujas execuções redundam em repetições da execução de uma determinada sequência de comandos.

4.2 O comando *for*

O comando *for* é uma estrutura de repetição que repete a execução de uma dada sequência de comandos um número de vezes que pode ser determinado pelo próprio programa, devendo ser escrito com a seguinte sintaxe:

```
for (inicializações; condições de manutenção da repetição; incrementos)
  {
  sequência de comandos
  }
```

Como os nomes indicam, em *inicializações*, são atribuídos valores iniciais a variáveis; em *condições de manutenção da repetição*, estabelecem-se, através de uma expressão, as condições nas quais a execução da sequência de comandos será repetida; em *incrementos*, incrementam-se variáveis. Quando um comando *for* é executado, a sequência de comandos da área das *inicializações* é executada. Em seguida, a expressão que fixa as *condições de manutenção da repetição* é avaliada. Se o valor desta expressão não for nulo, a sequência de comandos é executada, sendo em seguida executada a sequência de comandos da área dos *incrementos*. Novamente a expressão das *condições de manutenção da repetição* é avaliada e tudo se repete até que o seu valor seja igual a zero.

Por exemplo, o programa

```
#include <stdio.h>
main()
{
int i;
for (i = 1; i <= 10; i = i + 1)
        printf("%d    ", i);
}
```

exibe na tela os números 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. Por seu turno, o programa

```
#include <stdio.h>
main()
{
int i;
for (i = 10; i >= 0; i = i - 2)
        printf("%d    ", i);
}
```

exibe na tela os números 10, 8, 6, 4, 2, 0. Já o programa

```
#include <stdio.h>
main()
{
int i;
for (i = 1; i <= 10; i = i + 20)
        printf("%d    ", i);
}
```

exibe, apenas, o número 1.

A semântica do comando *for* implica que a sequência de comandos pode não ser executada nem uma única vez. Basta que na "primeira" execução do comando *for* a expressão que controla a repetição assuma o valor zero. Por exemplo, o programa abaixo não exibe nenhum valor na tela.

```
#include <stdio.h>
main()
```

```
{
int i;
for (i = 11; i <= 10; i = i + 20)
        printf("%d    ", i);
}
```

Com o comando *for*, a questão da geração de uma tabela de conversão de temperaturas em graus Fahrenheit para graus Celsius seria simples.

```
#include <stdio.h>
#include <conio.h>
main()
{
int  Fahrenheit;
float Celsius;
clrscr();
printf("Tabela de conversão graus Fahrenheit/graus Celsius \n");
printf("-------------------------------------------------\n");
printf("\t Fahrenheit \t | \t Celsius\n");
printf("-------------------------------------------------\n");
for (Fahrenheit = 20; Fahrenheit <= 80; Fahrenheit = Fahrenheit + 1)
        {
        Celsius = 5.0*(Fahrenheit - 32)/9;
        printf("\t %.2f \t | \t %.2f \n", Fahrenheit, Celsius);
        }
}
```

Na execução do comando *for*, a variável *Fahrenheit* é inicializada com o valor 20, este valor é comparado com 80, a correspondente temperatura em graus Celsius é calculada e os dois valores são exibidos. Em seguida, o conteúdo de *Fahrenheit* é incrementado de uma unidade e tudo se repete até que *Fahrenheit* atinja o valor 81. Desta forma, a execução deste programa gera a seguinte tabela

Tabela de conversão graus Fahrenheit/graus Celsius

Fahrenheit	Celsius
20	-6.67
21	-5.11
22	-5.56
23	-5.00
. . .	. . .
79	26,11
80	26,67

Observe que toda a repetição é controlada pela variável *Fahrenheit*. Num caso como este, a variável em foco é chamada *variável de controle* da estrutura de repetição. Vale observar também que, ao contrário de outras linguagens (Pascal, por exemplo), a variável de controle não tem que ser necessariamente do tipo *int*. Por exemplo, se quiséssemos que a tabela também fornecesse temperaturas em graus Fahrenheit fracionárias (meio em meio grau, por exemplo), poderíamos executar o seguinte programa.

```
#include <stdio.h>
#include <conio.h>
main()
{
float Celsius, Fahrenheit;
clrscr();
printf("Tabela de conversão graus Fahrenheit/graus Celsius \n");
printf("-------------------------------------------------\n");
printf("\t Fahrenheit \t | \t Celsius\n");
printf("-------------------------------------------------\n");
for (Fahrenheit = 20; Fahrenheit <= 80; Fahrenheit = Fahrenheit + 0.5)
        {
```

```
        Celsius = 5.0*(Fahrenheit - 32)/9;
        printf("\t %.2f \t | \t %.2f \n", Fahrenheit, Celsius);
        }
}
```

Cabe observar que a sequência de comandos cuja execução se pretende repetir pode ser colocada na área dos *incrementos*. O programa acima ficaria então com a seguinte forma:

```
#include <stdio.h>
#include <conio.h>
main()
{
float Celsius, Fahrenheit;
clrscr();
printf("Tabela de conversão graus Fahrenheit/graus Celsius \n");
printf("-------------------------------------------------\n");
printf("\t Fahrenheit \t | \t Celsius\n");
printf("-------------------------------------------------\n");
for (Fahrenheit = 20; Fahrenheit <= 80;  Celsius = 5.0*(Fahrenheit – 32)/9, printf("\t %.2f \t | \t %.2f
\n", Fahrenheit, Celsius), Fahrenheit = Fahrenheit + 0.5);
}
```

Observe que, neste caso, o comando *for* é concluído com um ponto-e-vírgula e que a leitura do programa fica bastante dificultada. Este fato faz com que esta prática não seja incentivada ao longo do livro.

Observe também que aproveitamos a oportunidade para apresentar mais uma função de biblioteca do sistema; trata-se de *clrscr()* cuja execução resulta na limpeza da *janela do usuário*, o que evita que resultados da execução de um programa sejam confundidos com resultados de execuções anteriores (*clrscr* vem de *clear screen* que significa "limpa tela"). Como indicado na segunda instrução do programa, o cabeçalho da função *clrscr()* encontra-se no arquivo *conio.h*.

4.3 O comando *while*

Para introduzir uma nova estrutura de repetição e cotejá-la com o comando *for*, considere um programa para encontrar um *divisor próprio* de um inteiro dado (um *divisor próprio* de um inteiro *n* é um divisor de *n* diferente dele e de 1). Esta questão é importante na verificação da *primalidade* de um inteiro: um número que não tem divisores próprios é dito *primo*. Com a utilização do comando *for* teríamos a seguinte solução para esta questão.

```
/*Programa que determina um divisor próprio de um inteiro */
#include <stdio.h>
main()
{
int Num, i, Divisor;
printf("Digite um numero: ");
scanf("%d", &Num);
Divisor = 0;
for (i = 2; i < Num; i = i + 1)
        if (Num % i == 0)
                Divisor = i;
if (Divisor != 0)
        printf("%d é divisor próprio de %d \n", Divisor, Num);
else
        printf("%d não tem divisores próprios \n", Num);
}
```

Um problema com este programa é que ele retorna sempre, se existir, o maior divisor próprio. Isto significa que se a entrada for um número par a estrutura de repetição não é interrompida quando o divisor 2 é

encontrado, o que, evidentemente, vai prejudicar a performance do programa. Isto pode ser contornado pois os compiladores C permitem que uma variável de controle de um comando *for* tenha o seu conteúdo alterado dentro do próprio comando. Com isto, o programa acima ficaria da seguinte forma.

```
#include <stdio.h>
main()
{
int Num, i, Divisor;
printf("Digite um número inteiro: ");
scanf("%d", &Num);
Divisor = 0;
for (i = 2; i < Num; i = i + 1)
	if (Num % i == 0)
		{
		Divisor = i;
		i = Num;
		}
if (Divisor != 0)
	printf("%d e' divisor próprio de %d \n", Divisor, Num);
else
	printf("%d não tem divisores próprios \n", Num);
}
```

Nesta versão, quando o primeiro divisor próprio é encontrado, o comando *i = Num;* faz com que a execução do comando *for* seja interrompida. A prática de encerrar um comando *for* através da alteração do conteúdo da variável de controle não será aqui incentivada pelo fato de que isto *desestrutura* o programa, dificultando sua legibilidade. Além disso, há situações em que não se pode conhecer o número máximo de repetições de uma estrutura de repetição. Na verdade, a questão central é que o comando *for* deve ser utilizado quando o número de repetições de execução de uma sequência de comandos é conhecido *a priori*. Quando isto não acontece (que é o caso do exemplo anterior: não se sabe *a priori* se e quando um divisor próprio vai ser encontrado), deve-se usar o comando *while*, que possui a seguinte sintaxe:

```
while (Expressão)
   {
   Sequência de comandos
   }
```

sendo os delimitadores opcionais se a sequência possui um só comando (como acontece nas outras estruturas de repetição).

A semântica deste comando é óbvia: a sequência de comandos é executada enquanto o valor da *Expressão* for diferente de zero. Naturalmente, pode ocorrer que a sequência de comandos não seja executada nenhuma vez, isto ocorrendo se o valor da *Expressão* for igual a zero quando da "primeira" execução do comando (o *teste* é feito antes da execução da sequência de comandos). Por outro lado, é necessário que um dos comandos da sequência de comandos altere conteúdos de variáveis que aparecem na *Expressão* de modo que em algum instante ela se torne igual a zero. Do contrário, a sequência de comandos terá sua execução repetida indefinidamente, o programa nunca termina e, evidentemente, não executa a tarefa para a qual foi desenvolvido. Quando isto acontece é comum se dizer que o programa está em *looping*.

Com o comando *while* as questões levantadas acima sobre o programa para determinar um divisor próprio de um inteiro dado são resolvidas e temos o seguinte programa:

```
/*Programa que determina o menor divisor próprio de um inteiro */
#include <stdio.h>
#include <conio.h>
main()
{
int Num, d, Met;
printf("Digite o numero: ");
scanf("%d", &Num);
Met = Num/2;
d = 2;
```

```
while (Num % d != 0 && d < Met)
  d++;
if (d <= Met)
  printf("%d  é divisor de %d \n", d, Num);
else
  printf("%d não tem divisores próprios", Num);
getch();
}
```

Observe que, ao contrário dos exemplos anteriores, a estrutura também seria interrompida quando a variável com a qual se procura um divisor atingisse a "metade" do inteiro; isto se explica pelo fato de que se um inteiro não possui um divisor próprio menor do que sua "metade", então ele é primo. Esta versão ainda pode ser melhorada utilizando-se o fato discutido em [Evaristo, J 2002] de que se um inteiro não possui um divisor próprio menor do que ou igual a sua raiz quadrada, ele não tem divisores próprios. Levando isso em conta, teríamos o seguinte programa.

```
/*Programa que determina o menor divisor próprio de um inteiro*/
#include <stdio.h>
#include <conio.h>
#include <math.h>
main()
{
int Num, d;
float r;
printf("Digite o numero: ");
scanf("%d", &Num);
r = sqrt(Num);
d = 2;
while (Num % d != 0 && d <= r)
  d++;
if (d <= r)
        printf("%d  é divisor de %d \n", d, Num);
else
  printf("%d não tem divisores próprios", Num);
getch();
}
```

Como já foi dito, um número inteiro que não tem divisores próprios é chamado *número primo*. Assim, o comando de saída vinculado à opção *else* poderia ser

```
printf("%d é primo", Num);
```

Vale observar que o comando d = 2; dos programas acima atribuiu um *valor inicial* à variável *d*. Este valor é incrementado de uma unidade enquanto um divisor não foi encontrado. Um comando de atribuição de um valor inicial a uma variável é chamado *inicialização da variável* e os compiladores da linguagem C permitem que inicializações de variáveis sejam feitas no instante em que elas são declaradas. Assim, as declarações de variáveis dos programas acima poderiam ter sido feitas da seguinte forma:

```
int Num, i, d = 2;
```

Neste livro, na maioria das vezes vamos optar por inicializar as variáveis imediatamente antes da necessidade. A razão desta opção é que há situações, como mostraremos no próximo exemplo, em que não se pode simplesmente inicializar uma variável quando da sua declaração.

Observe que o último comando dos últimos dois programas foi uma chamada da função *getch()*. Como já foi dito, a execução desta função requer a digitação de alguma tecla. Isto faz com a janela do usuário (que exibe o resultado do processamento) permaneça ativa até que uma tecla seja acionada.

Repetindo a execução de um programa

Uma outra aplicação importante do comando *while* diz respeito a aplicações sucessivas de um programa. O leitor deve ter observado que os programas anteriores são executados apenas para uma entrada. Se quisermos a sua execução para outra entrada precisamos executar o programa de novo.

Pode-se repetir a execução de um programa quantas vezes se queira, colocando-o numa estrutura definida por um comando *while*, controlada pelo valor de algum dado de entrada. Neste caso, o valor que encerra a execução pode ser informado dentro da mensagem que indica a necessidade da digitação da entrada. O programa anterior poderia ser então escrito da seguinte forma.

```
/*Programa que determina o menor divisor próprio de um inteiro */
#include <stdio.h>
#include <conio.h>
#include <math.h>
main()
{
int Num, d;
float r;
printf("Digite o numero (zero para encerrar): ");
Num = 1;
while (Num != 0)
        {
        scanf("%d", &Num);
        r = sqrt(Num);
        d = 2;
        while (Num % d != 0 && d <= r)
            d++;
        if (d <= r)
            printf("%d  é divisor de %d \n", d, Num);
        else
            printf("%d é primo", Num);
        }
}
```

Observe que, neste caso, a variável *d* não pode ser inicializada quando da sua declaração. Observe também que não há necessidade da função *getch()*, pois a própria repetição da execução deixa a janela do usuário aberta.

Alguns programadores preferem que a repetição da execução de um programa seja determinada por uma pergunta ao usuário do tipo “Deseja continuar (S/N)?”. Neste caso, há necessidade de uma variável do tipo *char* para receber a resposta e controlar a repetição da execução do programa.

```
#include <stdio.h>
#include <ctype.h>
#include <conio.h>
#include <math.h>
main()
{
int Num, d;
float r;
char c;
c = 'S';
while (toupper(c) == 'S')
        {
        printf("Digite o numero: ");
        scanf("%d", &Num);
        r = sqrt(Num);
        d = 2;
```

```
        while (Num % d != 0 && d <= r)
            d++;
        if (d <= r)
            printf("%d  é divisor de %d \n", d, Num);
        else
            printf("%d é primo", Num);
        puts("Deseja continuar (S/N)?");
        c = getch();
        }
}
```

Vale lembrar que a função *toupper()* retorna o argumento no formato maiusculo. Esta função foi ativada aqui para que o usuário não se preocupe em digitar como resposta letras maiusculas. Qualquer letra que for digitada, a função a torna maiuscula e o sistema a compara com S (maiusculo).

4.4 O comando *do while*

Como dissemos na seção anterior, o número de execuções da sequência de comandos associada a um comando *while* pode ser zero. Há situações onde é importante se garantir a execução de uma sequência de comandos pelo menos uma vez. Uma situação onde isto é importante é a verificação da *consistência dos dados de entrada*. Esta ação consiste em se dotar o programa de recursos para recusar dados incompatíveis com a entrada do programa, só "recebendo" dados que satisfaçam às especificações (lógicas ou estabelecidas) dos dados de entrada. Por exemplo, se a entrada é um número correspondente a um mês do ano, o programa não deve aceitar uma entrada que seja menor do que 1 nem maior do que 12. Uma solução para esta questão utilizando o comando *while* poderia ser a seguinte:

```
int Mes;
printf("Digite o mês: ");
scanf("%d", &Mes);
while ((Mes < 1) || (Mes > 12))
        {
        printf("\a Digitacao errada! Digite de novo \n");
        printf("Digite o mês: ");
        scanf("%d", &Mes);
        }
```

Observe que, como a verificação da condição de repetição é feita no "início" do comando, há a necessidade de uma leitura antes da estrutura e outra dentro dela (só para lembrar, \a emite um *beep*).

O comando *do while* define uma estrutura de repetição que garante que uma sequência de comandos seja executada pelo menos uma vez. Sua sintaxe é:

```
do
        {
        Sequência de comandos;
        }
while (Expressao);
```

e sua semântica é a seguinte: a sequência de comandos é executada e a *Expressão* é avaliada; se o valor da *Expressão* for diferente de zero, a sequência de comandos é novamente executada e tudo se repete; do contrário, o comando que segue a estrutura é executado. É importante observar a necessidade do ponto-e-vírgula encerrando o *do while*.

A consistência da entrada de um dado relativo a um mês utilizando um comando *do while* poderia ser a seguinte.

```
int Mes;
do
        {
        printf("Digite mes: ");
```

```
        scanf("%d", &Mes);
        if  ((Mes < 1) || (Mes > 12))
            printf("\a Digitacao errada! Digite de novo \n");
        }
while ((Mes < 1) || (Mes > 12));
```

A utilização do comando *do while* para execuções sucessivas de um programa é mais natural, quando a repetição da execução é feita através da resposta à pergunta *Deseja continuar (S/N)?* . Teríamos algo como:

```
#include <stdio.h>
main()
{
char Resp;
do
  {
        Sequência de comandos do programa propriamente dito;
        printf("Deseja continuar (S/N)?");
        scanf("%c", &Resp);
        }
while (toupper(Resp) == 'S');
}
```

4.5 O comando *break* em estruturas de repetição

Da mesma forma que sua ativação num *case* interrompe a execução de uma estrutura *switch*, a execução de um comando *break* dentro de uma estrutura de repetição interrompe as execuções da sequência de comandos da estrutura, mesmo que a condição de manutenção da repetição não tenha sido negada. Com o uso do *break*, o programa acima que determinava o menor divisor próprio de um inteiro poderia ter a seguinte forma:

```
#include <stdio.h>
#include <math.h>
main()
{
float r;
int  Num, d;
printf("Digite um numero : ");
scanf("%d", &Num);
d = 2;
r = sqrt(Num);
while (d <= r)
  if (Num % d == 0)
            break;
  else
            d = d + 1;
if (d <= r)
  printf("%d e' divisor proprio de %d \n", d, Num);
else
        printf("%d e' primo \n", Num);
}
```

Neste livro, o uso do *break* em estruturas de repetição não será estimulado visto que sua utilização pode trazer problemas de *legibilidade* aos programas.

4.6 Exemplos Parte IV

1. Consideremos um programa para determinar a soma dos n primeiros números ímpares, n dado. Por exemplo, se for fornecido para *n* o valor 6, o programa deve retornar 36, pois 1 + 3 + 5 + 7 + 9 + 11 = 36. Naturalmente, o sistema pode gerar os números impares que se pretende somar, através do comando *Impar = 1* e da repetição do comando *Impar = Impar + 2*. Naturalmente, também, para que o sistema gere o próximo ímpar, o anterior já deve ter sido somado. Isto pode ser feito através do comando *Soma = 0* e da repetição do comando *Soma = Soma + Impar*. Temos então o seguinte programa.

```
/*Programa que soma os n primeiros números ímpar, n dado*/
#include <stdio.h>
main()
{
int Soma, Impar, n, i;
printf("Digite o valor de n: ");
scanf("%d", &n);
Impar = 1;
Soma = 0;
for (i = 1; i <= n; i = i + 1)
        {
        Soma = Soma + Impar;
        Impar = Impar + 2;
        }
printf("Soma dos %d primeiros números impares: %d \n", n, Soma);
}
```

Observe que os comandos *Impar = 1* e *Soma = 0* atribuem um valor inicial às variáveis para que estes valores iniciais possam ser utilizados nas primeiras execuções dos comandos *Soma = Soma + Impar* e *Impar = Impar + 2*. Como já dissemos, nos referimos a comandos que atribuem valores iniciais a variáveis para que estes valores possam ser utilizados na primeira execução de um comando que terá sua execução repetida como *inicialização da variável.*

Uma outra observação interessante é que, como existe uma fórmula que dá o i-ésimo número ímpar ($a_i = 2i - 1$), o programa acima poderia ser escrito de uma forma mais elegante, prescindindo, inclusive, da variável Impar.

```
/*Programa que soma os n primeiros números impar, n dado*/
#include <stdio.h>
main()
{
int Soma, n, i;
printf("Digite o valor de n: ");
scanf("%d", &n);
Soma = 0;
for (i = 1; i <= n; i = i + 1)
        Soma = Soma + 2*i - 1;
printf("Soma dos %d primeiros números impares: %d \n", n, Soma);
}
```

Optamos por apresentar a primeira versão pelo fato de que nem sempre a fórmula para gerar os termos da sequência que se pretende somar é tão simples ou é muito conhecida. Por exemplo, o exercício número 2 da seção 4.7 pede para somar os quadrados dos *n* primeiros números naturais e, neste caso, embora a fórmula exista, ela não é tão conhecida.

2. Um dos exemplos da seção anterior apresentava um programa que determinava, se existisse, um divisor próprio de um inteiro dado. Imaginemos agora que queiramos um programa que apresente a lista de todos os divisores de um inteiro *n* dado. Neste caso, o programa pode percorrer todos os inteiros desde um até a metade de *n* verificando se cada um deles é um seu divisor. Temos então o seguinte programa.

```
#include <stdio.h>
```

```
main()
{
int Num, i;
printf("Digite o numero: ");
scanf("%d", &Num);
printf("Divisores próprios de %d: \n", Num);
for (i = 2; i <= Num/2; i = i + 1)
        if (Num % i == 0)
          printf("%d \n", i);
}
```

Vale observar que, ao contrário do que foi dito na seção 2.9, os valores de saída deste programa não estão sendo armazenados. O que acontece é que ainda não temos condições de armazenar uma quantidade indefinida de elementos. Este problema será resolvido no capítulo 6.

3. Na seção 1.5 discutimos um algoritmo que determinava o quociente e o resto da divisão entre dois inteiros positivos dados. Embora os compiladores de C possuam o operador % que calcula o resto de uma divisão inteira entre dois inteiros positivos, vamos apresentar, por ser interessante, a implementação do algoritmo referido.

```
/*Programa que determina o quociente e o resto da divisão entre dois inteiros positivos*/
#include <stdio.h>
main()
{
int Dividendo, Divisor, Quoc, Resto;
printf("Digite o dividendo e o divisor (diferente de zero!): ");
scanf("%d %d", &Dividendo, &Divisor);
Quoc = 1;
while (Quoc * Divisor <= Dividendo)
        Quoc = Quoc + 1;
Quoc = Quoc - 1;
Resto = Dividendo - Quoc * Divisor;
printf("Quociente e resto da divisão de %d por %d: %d e %d\n", Dividendo, Divisor, Quoc, Resto);
}
```

4. Em muitos casos há necessidade de que um dos comandos da sequência que terá sua execução repetida através de uma estrutura de repetição seja uma outra estrutura de repetição (num caso deste dizemos que as estruturas estão *aninhadas*). Para um exemplo, sejam A = {1, 2, 3, ..., n} e um programa que pretenda exibir o *produto cartesiano* AxA. Observe que para cada valor da primeira componente o programa deve gerar todas as segundas componentes. Devemos ter, portanto, uma estrutura de repetição para gerar as primeiras componentes e uma outra, vinculada a cada valor da primeira componente, para gerar as segundas componentes.

```
/*Programa para gerar um produto cartesiano*/
#include <stdio.h>
main()
{
int n, i, j;
printf("Digite o numero de elementos do conjunto: ");
scanf("%d", &n);
printf("{");
for (i = 1; i <= n; i = i + 1)
        for (j = 1; j <= n; j = j + 1)
            printf("(%d, %d), ", i, j);
printf("}");
}
```

5. É interessante observar que a variável de controle da estrutura *interna* pode depender da variável de controle da estrutura *externa*. Por exemplo, se ao invés dos pares ordenados, quiséssemos os subconjuntos do conjunto A com dois elementos, o programa não deveria exibir o subconjunto {1, 1}, que possui um só elemento, e deveria exibir apenas um dos subconjuntos {1, 2} e {2, 1} já que eles são iguais. Isto pode ser

obtido inicializando j com uma unidade maior do que o valor de i.

```
/*Programa para gerar um conjunto de subconjuntos de um conjunto*/
#include <stdio.h>
main()
{
int n, i, j;
printf("Digite o numero de elementos do conjunto: ");
scanf("%d", &n);
printf("{");
for (i = 1; i <= n; i = i + 1)
        for (j = i + 1; j <= n; j = j + 1)
            printf("{%d, %d}, ", i, j);
printf("}");
}
```

6. Seja um programa para o cálculo da média de uma dada quantidade de números. Na seção 1.5 discutimos um algoritmo para determinar a média de 10.000 números dados. Na ocasião discutimos que utilizaríamos uma única variável para receber os números sendo que um valor subsequente só seria solicitado depois que o anterior fosse "processado". A diferença agora é que a quantidade de números será um dado de entrada, o que torna o programa de aplicação mais variada. Como a quantidade de números será dada, pode-se utilizar uma estrutura *for* para receber e somar os números.

```
/*Programa para calcular a media de n numeros, n dado*/
#include <stdio.h>
main()
{
int n, i;
float Num, Soma, Media;
Soma = 0;
printf("Digite o numero de elementos: ");
scanf("%d", &n);
printf("\n Digite os elementos:");
for (i = 1; i <= n; i = i + 1)
        {
        scanf("%f", &Num);
        Soma = Soma + Num;
        }
Media = Soma/n;
printf("Media = %f", Media);
}
```

7. O exemplo acima tem o inconveniente de que sua execução exige que se saiba anteriormente a quantidade de números e isto não ocorre na maioria dos casos. Vejamos então um programa para determinar a média de uma relação de números dados, sem que se conheça previamente a quantidade deles. Neste caso, não devemos utilizar o comando *for*, pois não sabemos o número de repetições! Assim, o comando *while* deve ser utilizado; porém, uma pergunta deve ser formulada: qual a expressão lógica que controlará a estrutura? A solução é "acrescentar" à relação um valor sabidamente diferente dos valores da relação e utilizar este valor para controlar a repetição. Este valor é conhecido como *flag*. Como dito logo acima, deve-se ter certeza que o *flag* não consta da relação. Isto não é complicado, pois ao se escrever um programa se tem conhecimento de que valores o programa vai manipular e a escolha do *flag* fica facilitada. Por exemplo, se o programa vai manipular números positivos pode-se usar -1 para o *flag*. Além do *flag*, o programa necessita de uma variável (no caso *Cont* de contador) que determine a quantidade de números da relação, pois este valor será utilizado no cálculo da média.

```
/*Programa para calcular a media de uma relacao de numeros*/
#include <stdio.h>
main()
{
int Cont;
```

```
float Num, Soma, Media;
Soma = 0;
printf("\n Digite os elementos(-1 para encerrar):");
scanf("%d", &Num);
Cont = 0;
while (Num != -1)
        {
        Soma = Soma + Num;
        Cont = Cont + 1;
        scanf("%f", &Num);
        }
Media = Soma/Cont;
printf("Media = %f", Media);
}
```

8. Na seção 1.6 apresentamos o *algoritmo de Euclides* para a determinação do *máximo divisor comum* de dois números dados. Para relembrar, vejamos como calcular o máximo divisor comum de 204 e 84.

	2	2	3
204	84	36	12

O algoritmo é o seguinte: divide-se 204 por 84 obtendo-se resto 36; a partir daí repete-se divisões até que o resto seja zero, sendo o dividendo da divisão atual o divisor da divisão anterior e o divisor da divisão atual o resto da divisão anterior. O último divisor é o máximo divisor procurado.

Escrever este algoritmo numa linguagem de programação é muito simples, pois uma estrutura de repetição e comandos de atribuição permitem que se obtenha facilmente a sequência de divisões desejadas, “atualizando” o dividendo, o divisor e o resto.

```
/*Programa para determinar o máximo divisor comum de dois números positivos*/
#include <stdio.h>
main()
{
int x, y, Dividendo, Divisor, Mdc, Resto;
printf("Digite os dois numeros  \n");
scanf("%d %d", &x, &y);
Dividendo = x;
Divisor = y;
Resto = Dividendo % Divisor;
while (Resto != 0)
        {
        Dividendo = Divisor;
        Divisor = Resto;
        Resto = Dividendo % Divisor;
        }
Mdc = Dividendo;
printf("mdc(%d, %d) = %d \n", x, y, Mdc);
}
```

Note a necessidade da utilização das variáveis *Dividendo* e *Divisor*. Além de facilitarem a compreensão do algoritmo, elas são utilizadas no processamento e terão seus conteúdos alterados durante a execução do programa. Se usássemos as variáveis x e y, os valores dos dados de entrada seriam perdidos o que, evidentemente, não deve ocorrer. No capítulo 5, quando estudarmos funções, estas variáveis terão outra conotação.

À primeira vista, o programa deveria inicialmente determinar o maior dos números x e y, armazenando-o em a. O quadro seguinte mostra que isto não é necessário, apresentando a simulação da execução do programa para x = 68 e y = 148.

x	y	a	b	Resto	Mdc
68	148	68	148	68	
		148	68	12	
		68	12	8	
		12	8	4	
		8	4	0	
					4

9. Um outro algoritmo matemático cuja implementação numa linguagem de programação apresenta um bom exemplo do uso de estruturas de repetição é o algoritmo para a determinação do *mínimo múltiplo comum* (*mmc*) de dois números dados. Como indica a própria denominação, o *mínimo múltiplo comum* de dois números é o menor número que é divisível pelos dois números. A matemática prova que o *mmc* de dois números é o produto dos divisores primos dos dois números, comuns ou não, ambos com as suas multiplicidades. Para se obter os divisores primos, realiza-se divisões sucessivas pelos primos que são divisores de pelo menos um dos números.

A tabela seguinte mostra o cálculo do mínimo múltiplo comum dos números 360 e 420, como nos é ensinado no ensino fundamental.

```
360, 420|2
180, 210|2
 90, 105|2
 45, 105|3
 15,  35|3
  5,  35|5
  1,   7|7
  1,   1|MMC = 2*2*2*3*3*5*7 = 2 520
```

Observe que, quando um divisor primo é encontrado, repete-se a divisão, com o quociente no lugar do dividendo até se obter um número que não é múltiplo daquele divisor. Aí, incrementa-se o tal divisor. Isto é feito até que ambos os quocientes sejam iguais a 1. Temos o seguinte programa.

```
/*Programa para determinar o minimo multiplo comum de dois numeros positivos*/
#include <stdio.h>
main()
{
int x, y, d, a, b, i, Mmc;
printf("Digite os dois numeros \n");
scanf("%d %d", &x, &y);
a = x;
b = y;
Mmc = 1;
i = 2;
while ((a != 1) || (b != 1))
        {
        while ((a % i == 0) || (b % i == 0))
            {
            if (a % i == 0)
                    a = a/i;
            if (b % i == 0)
                    b = b/i;
            Mmc = Mmc * i;
            }
        i = i + 1;
        }
printf("mmc(%d, %d) = %d \n", x, y, Mmc);
}
```

10. A questão do *mínimo múltiplo comum* é muito interessante como exemplo para a aprendizagem de programação pelo fato de que podemos apresentar um outro algoritmo de compreensão bem mais simples

que o anterior. A ideia é a seguinte: x, 2x, 3x, etc. são múltiplos de x. Para se obter o mínimo múltiplo comum basta que se tome o primeiro destes números que seja múltiplo também de y.

```
/*Programa para determinar o mínimo múltiplo comum de dois números positivos*/
#include <stdio.h>
main()
{
int x, y, i, Mmc;
printf("Digite os dois numeros \n");
scanf("%d %d", &x, &y);
Mmc = x;
while (Mmc % y != 0)
	Mmc = Mmc + x;
printf("mmc(%d, %d) = %d \n", x, y, Mmc);
}
```

4.7 Exercícios propostos

1. Mostre a configuração da tela após a execução do programa

```
#include <stdio.h>
main()
{
int i, a, q, Termo;
for (i = 5; i > 0; i = i - 1)
	{
	a = i;
	q = 3;
	Termo = a;
	while (Termo <= 9 * a)
	    {
	    printf("%d \n", Termo);
	    Termo = Termo * q;
	    }
	}
}
```

2. Escreva um programa que determine a soma dos quadrados dos n primeiros números naturais, n dado.

3. Escreva um programa para calcular a soma dos n primeiros termos das sequências abaixo, n dado.

a) $\left(\frac{1}{2}, \frac{3}{5}, \frac{5}{8}, ...\right)$

b) $\left(1, -\frac{1}{2}, \frac{1}{3}, -\frac{1}{4}, ...\right)$

4. O exemplo 10 da seção anterior apresentava uma solução para a questão do *mínimo múltiplo comum* de simples compreensão. Um problema que esta solução possui é que se o primeiro valor digitado fosse muito menor do que o segundo, o número de repetições necessárias para se chegar ao *mmc* seria muito grande. Refaça o exemplo, tomando o maior dos números dados como base do raciocínio ali utilizado.

5. Um número inteiro é dito *perfeito* se o dobro dele é igual à soma de todos os seus divisores. Por exemplo, como os divisores de 6 são 1, 2, 3 e 6 e 1 + 2 + 3 + 6 = 12, 6 é perfeito. A matemática ainda não sabe se a quantidade de números perfeitos é ou não finita. Escreva um programa que liste todos os números perfeitos menores que um inteiro n dado.

6. O número 3.025 possui a seguinte característica: 30 + 25 = 55 e 55^2 = 3 025. Escreva um programa que escreva todos os números com quatro algarismos que possuem a citada característica.

7. Escreva um programa que escreva todos os pares de números de dois algarismos que apresentam a

seguinte propriedade: o produto dos números não se altera se os dígitos são invertidos. Por exemplo, 93x13 = 39x31 = 1.209.

8. Escreva um programa para determinar o número de algarismos de um número inteiro positivo dado.

9. Um número inteiro positivo é dito *semiprimo* se ele é igual ao produto de dois números primos. Por exemplo, 15 é semiprimo pois 15 = 3 x 5; 9 é semiprimo pois 9 = 3 x 3; 20 não é semiprimo pois 20 = 2 x 10 e 10 não é primo. Os números semiprimos são fundamentais para o sistema de criptografia RSA {Evaristo, J, 2002]. Escreva um programa que verifique se um inteiro dado é semiprimo.

10. Quando um número não é semiprimo, a Matemática prova que ele pode ser escrito de maneira única como um produto de potências de números primos distintos. Este produto é chamado de *decomposição em fatores primos* do número e os expoentes são chamados de *multiplicidade* do primo respectivo. Por exemplo, $360 = 2^3 x 3^2 x 5$. Escreva um programa que obtenha a decomposição em fatores primos de um inteiro dado.

11. Escreva um programa que transforme o computador numa *urna eletrônica* para eleição, em segundo turno, para presidente de um certo país, às quais concorrem os candidatos 83-Alibabá e 93-Alcapone. Cada voto deve ser dado pelo número do candidato, permitindo-se ainda o voto 00 para *voto em branco*. Qualquer voto diferente dos já citados é considerado *nulo*; em qualquer situação, o eleitor deve ser consultado quanto à confirmação do seu voto. No final da eleição o programa deve emitir um relatório contendo a votação de cada candidato, a quantidade votos em branco, a quantidade de votos nulos e o candidato eleito.

12. A *sequência de Fibbonaci* é a sequência (1, 1, 2, 3, 5, 8, 13, ...) definida por

$$a_n = \begin{cases} 1, se\ n = 1\ ou\ n = 2 \\ a_{n-1} + a_{n-2}, se\ n > 2 \end{cases}$$

Escreva um programa que determine o n-ésimo termo desta sequência, n dado.

13. A *série harmônica* $S = 1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n} + ...$ é *divergente*. Isto significa que dado qualquer real k existe n_0 tal que $1 + \frac{1}{2} + \frac{1}{3} + ... + \frac{1}{n_0} > k$. Escreva um programa que dado um real k determine o menor inteiro n_0 tal que S > k. Por exemplo se k = 2, o programa deve fornecer n_0 = 4, pois $1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} = 2{,}083....$ e $1 + \frac{1}{2} + \frac{1}{3} = 1{,}8333....$

14. Dois números inteiros são ditos *amigos* se a soma dos divisores de cada um deles (menores que eles) é igual ao outro. Por exemplo, os divisores de 220 são 1, 2, 4, 5, 10, 11, 20, 22, 44, 55 e 110 e 1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 = 284 e os divisores de 284 são 1, 2, 4, 71 e 142 e 1 + 2 + 4 + 71 + 142 = 220. Escreva um programa que determine todos os pares de inteiros amigos menores que um inteiro dado.

15. Escreva um programa que escreva todos os subconjuntos com três elementos do conjunto {1, 2, 3, ..., n}, *n* dado.

16. Um inteiro positivo x é dito uma *potência prima* se existem dois inteiros positivos p e k, com p primo, tais que $x = p^k$. Escreva uma função que receba um inteiro e verifique se ele é uma potência prima.

17. Um inteiro positivo x é dito uma *potência perfeita de base z e expoente y* se existem dois inteiros positivos z e y tais que $x = z^y$. Escreva uma função que receba um inteiro e verifique se ele é uma potência perfeita.

Observação

Propostas de soluções dos exercícios propostos podem ser solicitadas através de mensagem eletrônica para jaime@ccen.ufal.br com assunto RESPOSTAS LIVRO C, anexando o formulário abaixo devidamente preenchido.

Nome	Categoria[1]	Instituição[2]	Curso[2]	Cidade/Estado

[1]Categoria: docente, estudante, autodidata

[2]Se docente ou estudante

5. Funções e ponteiros

5.1 O que são funções

Como dissemos no capítulo 2, um programa em C pode (e deve) ser escrito como um conjunto de *funções* que são executadas a partir da execução de uma função denominada *main()*. Cada função pode conter declarações de variáveis, instruções, ativações de funções do sistema e de outras funções definidas pelo programador. Naturalmente, o objetivo de uma função deve ser a realização de alguma "sub-tarefa" específica da tarefa que o programa pretende realizar. Assim, pode-se escrever funções para a leitura dos dados de entrada, para a saída do programa, para a determinação da média de vários elementos, para a troca dos conteúdos de uma variável, para o cálculo do máximo divisor comum de dois números dados, etc. Normalmente, a realização da "sub-tarefa" para a qual a função foi escrita é chamada de *retorno* da função. Este retorno pode ser a realização de uma ação genérica, como a leitura dos dados de entrada, ou um valor específico, como o cálculo do máximo divisor comum de dois números dados.

Como foi dito na seção citada, uma função deve ser definida com a seguinte sintaxe:

```
Tipo de Dado Identificador da função(Lista de parâmetros)
  {
  Declaração de variáveis
  Sequência de instruções
  }
```

onde, como também já foi dito, o conjunto *Tipo de Dado Identificador da função(Lista de parâmetros)* é chamado *protótipo* da função. Aí, *Tipo de dado* é *int*, *char*, *float*, ou um dos seus "múltiplos", quando a função deve retornar um valor específico e *void* se a função deve realizar uma ação genérica sem retornar um valor definido. Por seu turno, *Lista de parâmetros* é um conjunto de variáveis utilizadas para a função receber os valores para os quais a função deve ser executada; estes valores são chamados *argumentos*, que, comparando uma função com um programa, constituem os *dados de entrada* da função.

Na declaração de variáveis são declaradas as variáveis que as instruções da função vão manipular internamente. Estas variáveis (e os parâmetros da função) só são acessáveis pelas instruções da função e, por esta razão, são chamadas *variáveis locais* (na seção 5.5 são apresentados maiores detalhes).

Se a função deve retornar um valor, uma de suas instruções deve ter a seguinte sintaxe:

```
return (expressão);
```

sendo os parênteses facultativos. A semântica desta instrução é evidente: a expressão é avaliada e o seu valor é retornado à função que a ativou. Além disso (e isto agora não é evidente), a execução desta instrução interrompe a execução da função e o processamento retorna à função que ativou a função em discussão.

A ativação (ou *chamada*) da função por outra função se faz com a referência ao identificador da função seguido dos argumentos em relação aos quais se pretende executar a função. Por exemplo, o cálculo do máximo divisor comum de dois números dados, discutido no capítulo anterior, poderia ser reescrito da seguinte forma:

```
/*Função que retorna o máximo divisor comum de dois números positivos dados*/
int MaxDivCom(int x, int y)
{
int Resto;
Resto = x % y;
while (Resto != 0)
        {
        x = y;
        y = Resto;
        Resto = x % y;
        }
return (y);
}
```

```
/*Função principal: recebe os números e ativa a função MaxDivCom()*/
main()
{
int Mdc, a, b;
printf("Digite os dois inteiros");
scanf("%d %d", &a, &b);
Mdc = MaxDivCom(a, b);
printf("Mdc(%d, %d) = %d \n", a, b, Mdc);
}
```

Se este programa for executado para a entrada a = 204 e b = 184, a execução do comando

Mdc = MaxDivCom(a, b)

chamaria a execução da função, recebendo o parâmetro *x* o conteúdo de *a* e o parâmetro *y* o conteúdo de *b*. Ou seja, as instruções da função seriam executadas para a entrada x = 204 e y = 184 e teríamos a seguinte sequência de valores para as variáveis locais *x*, *y* e *Resto* até a interrupção da estrutura *while*:

x	y	Resto
204	184	20
184	20	4
20	4	0

Neste instante, a instrução *return (y)* é executada e o processamento retorna ao comando

Mdc = MaxDivCom(a, b)

e o valor 4 é armazenado na variável Mdc.

Em alguns sistemas, uma função só pode ativar uma outra função que foi definida previamente (como no exemplo anterior) ou cujo protótipo esteja explicitado como uma das suas instruções. Por exemplo, o programa aqui referido poderia ter o seguinte formato:

```
main()
{
int Mdc, a, b;
int MaxDivCom(int x, int y); /*Protótipo da função que vai ser definida posteriormente */
printf("Digite os dois inteiros");
scanf("%d %d", &a, &b);
Mdc = MaxDivCom(a, b);
printf("Mdc(%d, %d) = %d \n", a, b, Mdc);
}
int MaxDivCom(int x, int y)
{
int Resto;
Resto = x % y;
while (Resto != 0)
	{
	x = y;
	y = Resto;
	Resto = x % y;
	}
return (y);
}
```

5.2 Para que servem funções

Evidentemente, não há grandes vantagens do programa que calcula o máximo divisor comum escrito

com uma função em relação àquele do exemplo já citado, que realizava todas as ações necessárias dentro da função *main()*. Na verdade, a utilização de funções só é bastante vantajosa quando se trata de programas grandes, capazes de realizar diversas tarefas independentes, mas relacionadas. Por exemplo, um programa que gerencie as contas correntes de um banco deve ser capaz, entre outras coisas, de fornecer o saldo de uma dada conta; atualizar o saldo em função da ocorrência de uma retirada ou de um depósito; cadastrar uma nova conta; excluir do cadastro uma dada conta, etc. Naturalmente, embora estas tarefas estejam relacionadas, deve-se pretender que elas sejam realizadas de forma independente, pois um dado cliente num dado momento pode querer a realização de apenas uma delas.

Num caso como este, o programa deve possuir uma função para cada uma das tarefas pretendidas, ficando a cargo da função *main()* a chamada de uma ou de outra função de acordo com a tarefa pretendida. Isto permite que vários programadores desenvolvam o programa (cada um desenvolvendo algumas funções), facilita a realização de testes de correção do programa (as funções podem ser testadas de forma isolada) e a manutenção posterior do programa (modificações necessárias no programa podem ficar restritas a modificações em algumas das funções). Na verdade, a *modularização* do programa só traz benefícios e deve ser uma prática de todo programador.

É comum que um programa "multi-tarefa" como o exemplificado acima seja iniciado com a disponibilização para o usuário das diversas tarefas que ele é capaz de executar. Normalmente, este conjunto de tarefas é chamado de *menu de opções* e pode ser obtido através de uma função que não retorna nenhum valor. Considerando apenas as tarefas listadas acima, o menu de opções do programa referido poderia ser construído a partir da seguinte função.

```
void Menu()
{
printf("1-Saldo \n 2-Depósito \n 3-Retirada \n 4-Nova conta \n 5-Encerra conta \n 6-Sai do
programa);"
}
```

Neste caso, uma das primeiras instruções da função *main()* é a ativação da função *Menu()* com a simples referência ao seu identificador seguido de parênteses vazios e de ponto-e-vírgula:

```
main()
{
Menu();
. . .
}
```

Observe que esta função exemplifica uma função que não retorna um valor (daí ser do tipo void) e cuja *Lista de parâmetros* é vazia.

Atualmente, com a disponibilização das linguagens visuais (VisualBasic, Delphi e outras), os *menus de opções* são disponibilizados através de *interfaces* programa/usuário (contendo *botões*, *banners* e outras denominações) e as ativações das funções que executam a tarefa pretendida é feita através de *mouses* ou mesmo através de toque manual na tela do computador. Existem algumas *bibliotecas gráficas* que permitem que se criem *interfaces* de programas em C; porém, o estudo destas bibliotecas não está no escopo deste livro.

Outra aplicação importante de funções se dá quando há necessidade de que o programa determine a mesma grandeza para valores diferentes. Um exemplo típico desta necessidade aparece num programa que determine medidas estatísticas, como *média aritmética*, *mediana*, *desvio médio*, *desvio padrão*, de uma relação de números. Como o *desvio médio* é a média aritmética dos valores absolutos dos desvios em relação à média, o seu cálculo exigirá a determinação da média aritmética da relação e a média aritmética dos desvios. Escreveremos então uma função para o cálculo da média de uma relação qualquer e a utilizaremos para os cálculos das duas médias necessárias. Este exemplo será visto no capítulo seguinte.

5.3 Passagem de parâmetros

Uma outra possível utilização de funções é para substituir uma sequência de instruções que se repete em várias partes do programa. Por exemplo, o exemplo 3 da seção 3.4 apresentava um programa para ordenar três números:

```
/* Programa para ordenar tres numeros dados*/
#include <stdio.h>
main()
{
float x, y, z, Aux;
printf("Digite os tres numeros");
scanf("%f %f %f", &x, &y, &z);
printf("Numeros dados: %f , %f , %f \n", x, y, z);
if ((x > y) || (x > z))              /* verifica se x não é o menor */
        if (y < z)                              /* neste caso y é o menor */
            {
            Aux = x;                            /* troca os conteúdos de x e de y */
            x = y;
            y = Aux;
            }
        else                                            /* neste caso z é o menor */
            {
            Aux = x;                            /* troca os conteúdos de x e de z */
            x = z;
            z = Aux;
            }
if (y > z)                                      /* verifica se z e y ainda não estão ordenados */
        {
        Aux = y;                                /* troca o conteúdo de y e de z */
        y = z;
        z = Aux;
        }
printf("Numeros ordenados: %f , %f , %f \n", x, y, z);
}
```

Observe que uma sequência de comandos com o mesmo objetivo (trocar os conteúdos de duas variáveis) se repete. Num caso como este poderíamos escrever uma função que realizasse aquela ação pretendida e, então, esta função seria utilizada para substituir a sequência referida.

Por enquanto temos o seguinte problema. Se definirmos a função

```
void Troca(float x, float y)
  {
  float Aux;
  Aux = x;
  x = y;
  y = Aux;
  }
```

e a executarmos passando as variáveis *a* e *b*, apenas os conteúdos de *a* e de *b* serão passados para *x* e para *y* e a troca realizada pela função só afeta os conteúdos de *x* e de *y*, não modificando os conteúdos de *a* e de *b* que é o que se pretendia. Ou seja, a função *Troca* recebe apenas os "valores" de *a* e de *b* e as ações realizadas pela função interfere apenas nos parâmetros *x* e *y*, não alterando nada em relação aos argumentos *a* e *b*. Neste caso, dizemos que os parâmetros foram passados *por valor*.

O sistema Turbo C++ 3.0 oferece a possibilidade de que a execução de uma função altere conteúdos de variáveis “não locais”. Para isto, no protótipo da função os parâmetros devem ser precedidos de &. Neste caso, os argumentos para ativação da função têm que ser variáveis e qualquer alteração no conteúdo do parâmetro se reflete no conteúdo da variável argumento. Diz-se então que a passagem dos parâmetros é feita *por referência*.

Com este tipo de passagem de parâmetro, o programa acima poderia ser escrito da seguinte forma:

```
/* Programa para ordenar tres numeros dados*/
#include <stdio.h>
void Troca(float &a, float &b)
```

```
  {
  float Aux;
  Aux = a;
  a = b;
  b = Aux;
  }
main()
{
float x, y, z, Aux;
printf("Digite os tres numeros");
scanf("%f %f %f", &x, &y, &z);
printf("Numeros dados: %f , %f , %f \n", x, y, z);
if ((x > y) || (x > z))
        if (y < z)
                Troca(x, y);
        else
                Troca(x, z);
if (y > z)
        Troca(y, z);
printf("Numeros ordenados: %f , %f , %f \n", x, y, z);
}
```

A passagem de parâmetro por referência permite que entrada de dados seja feita através de uma função. Isto pode ser útil, por exemplo, em programas multitarefas em que o número de entradas pode variar de acordo com a tarefa pretendida. Para exemplificar apresentaremos um programa para tratar números complexos. Naturalmente, um programa com este objetivo deve estar apto a somar e multiplicar complexos, casos em que a entrada será dois números complexos, e a determinar o módulo e a forma polar de um complexo, quando entrada será apenas de um número. Além de exemplificar a entrada de dados através de uma função, o programa abaixo exemplifica um programa multitarefa "completo".

```
/*Programa para álgebra dos números complexos*/
#include <stdio.h>
#include <math.h>
void LeComplexo(float &a, float &b)
  {
  puts("Digite a parte real <enter> parte imaginaria");
  scanf("%f %f", &a, &b);
  }
float Modulo(float a, float b)
  {
  return sqrt(a*a + b*b);
  }
void Polar(float a, float b, float &c, float &d)
  {
  c = Modulo(a, b);
  d = asin(b/c);
  }
void Soma(float a, float b, float c, float d, float &e, float &f)
  {
  e = a + c;
  f = b + d;
  }
void Produto(float a, float b, float c, float d, float &e, float &f)
  {
  e = a*c - b*d;
```

```
  f = a*d + b*c;
  }
void Menu()
  {
  puts("1-Modulo \n 2- Forma polar \n 3-Soma \n 4-Produto \n 5-Encerra \n Digite sua opcao: ");
  }
main()
{
float x, y, z, w, t, u;
int Opc;
Menu();
scanf("%d", &Opc);
switch (Opc)
  {
  case 1:
        LeComplexo(x, y);
        z = Modulo(x, y);
        printf("|%.2f + %.2fi| = %.2f", x, y, z);
        break;
  case 2:
        LeComplexo(x, y);
        Polar(x, y, z, w);
        printf("%.2f + %.2fi = %.2f(cos%.2f + isen%.2f)", x, y, z, w, w);
        break;
  case 3:
        LeComplexo(x, y);
        LeComplexo(z, w);
        Soma(x, y, z, w, t, u);
        printf("(%.2f + %.2fi) + (%.2f + %.2fi) = %.2f + %.2fi", x, y, z, w, t, u);
              break;
  case 4:
        LeComplexo(x, y);
        LeComplexo(z, w);
        Produto(x, y, z, w, t, u);
        printf("(%.2f + %.2fi) + (%.2f + %.2fi) = %.2f + %.2fi", x, y, z, w, t, u);
        break;
  }
}
```

O exemplo a seguir melhora sobremaneira a legibilidade do programa (parte dele) que determina o dia da semana de uma data posterior ao ano de 1600 dada apresentado no capítulo 3. Lá precisávamos determinar o número de dias decorridos entre 01/01/1600 e a data dada. Vimos que precisávamos determinar, entre outras coisas: o número de dias já decorridos no ano da data dada (para isto precisávamos determinar se tal ano era bissexto e o número de dias 31 já ocorridos) e a quantidade de anos bissextos entre 1600 e o ano da data dada. A boa técnica de programação sugere que cada ação parcial do programa seja executada por uma função.

Temos então a seguinte proposta para um programa que determine o número de dias dias decorridos entre duas datas dadas (este programa é utilizado em aposentadorias: pela legislação atual (novembro de 2008) um trabalhador de uma empresa privada adquire o direito de se aposentar quando completa 35 anos de serviço, sendo este cálculo a partir da soma do número de dias trabalhados nas (possivelmente) várias empresas nas quais o interessado trabalhou).

```
/*Programa para determinar o número de dias entre duas datas dadas*/
        #include <conio.h>
#include <stdio.h>
#include <conio.h>
```

```
/*Verifica se um ano é bissexto (retorno: Sim-1/Nao-0)*/
int EhBissexto(int Ano)
{
return ((Ano % 4 == 0) && ((Ano % 100 != 0) || (Ano % 400 == 0)));
}
/*Retorna o número de dias 31 ocorridos até o mês dado*/
int NumDias31(int Mes)
{
if (Mes < 9)
	return Mes/2;
else
	return (Mes + 1)/2;
}
/*Retorna o número de dias de um ano até uma data dada*/
int NumDiasAteUmaData(int Dia, int Mes, int Ano)
{
int NumDias;
//Numero de dias considerando todos os meses com 30 dias
NumDias = 30*(Mes - 1);
//Acrescentando o número de dias 31 já ocorridos no ano e o número de dias do mês corrente
NumDias = NumDias + NumDias31(Mes) + Dia;
//Retificando o número de dias de fevereiro (se ele já ocorreu)
if (Mes > 2)
	if (EhBissexto(Ano))
		NumDias = NumDias - 1;
	else
		NumDias = NumDias - 2;
return NumDias;
}
/*Retorna o número de dias de uma após uma data dada*/
int NumDiasAposUmaData(int Dia, int Mes, int Ano)
{
if (EhBissexto(Ano))
	return 367 - NumDiasAteUmaData(Dia, Mes, Ano);
else
	return 366 - NumDiasAteUmaData(Dia, Mes, Ano);
}
/*Retorna o número de anos bissextos entre dois anos dados*/
int NumAnosBissextos(int Ano1, int Ano2)
{
int Aux, Mult4, Mult100, Mult400;
Aux = Ano2 - 1;
Mult4 = (Aux - (Aux % 4) - Ano1 + (Ano1 % 4))/4;
Mult100 = (Aux - (Aux % 100) - Ano1 + (Ano1 % 100))/100;
Mult400 = (Aux - (Aux % 400) - Ano1 + (Ano1 % 400))/400;
return Mult4 - Mult100 + Mult400;
}
main()
{
int Dia1, Mes1, Ano1, Dia2, Mes2, Ano2, Anos, NumDias, DiasDoAnoFinal, DiasDoAnoInicial;
clrscr();
printf("Data inicial (dd/mm/aaaa)\n");
scanf("%d/%d/%d", &Dia1, &Mes1, &Ano1);
printf("Data final (dd/mm/aaaa)\n");
scanf("%d/%d/%d", &Dia2, &Mes2, &Ano2);
```

```
Anos = Ano2 - Ano1 - 1;
DiasDoAnoFinal = NumDiasAteUmaData(Dia2, Mes2, Ano2);
DiasDoAnoInicial = NumDiasAposUmaData(Dia1, Mes1, Ano1);
NumDias = Anos*365 + DiasDoAnoFinal + DiasDoAnoInicial + NumAnosBissextos(Ano1, Ano2);
printf("\nData inicial: %d/%d/%dData final: %d/%d/%d Numeros de dias: %d", Dia1, Mes1, Ano1, Dia2, Mes2, Ano2, NumDias);
getch();
}
```

Embora o *help* do Turbo C++ 3.0 afirme que em C a única passagem de parâmetro é por valor, conseguimos, no Turbo C 2.01, uma forma de passagem de parâmetros por referência. Para isto utilizaremos *ponteiros* que além de permitir esta forma de passagem de parâmetros tem outras aplicações importantes em programação em C e em C++.

5.4 Ponteiros

No capítulo 2, foi dito que a cada posição de memória é associado um número inteiro chamado *endereço* da posição de memória. Como uma *variável* é uma posição de memória, a cada variável é associado um endereço.

Um *ponteiro* é uma variável capaz de armazenar um endereço de outra variável, sendo declarado com a seguinte sintaxe:

```
Tipo de dado *Identificador;
```

A semântica desta declaração pode ser assim entendida: *Identificador* é capaz de armazenar o endereço de uma variável de tipo *Tipo de dado*. Por exemplo, uma declaração do tipo

```
int *p;
```

indica que *p* é uma variável capaz de armazenar o endereço de uma variável do tipo *int*. Na prática dizemos que p *aponta* para um inteiro. Assim, ponteiros também são chamados *apontadores*. Como ponteiros são variáveis, pode-se atribuir um ponteiro a outro do mesmo tipo. Por exemplo, são válidas as seguintes instruções:

```
int *p, *t, i;
p = &i;
t = p;
```

pois, como dissemos no capítulo 2, o *operador de endereço* & fornece o endereço da variável em que ele está operando. Deste modo, *p* receberá o endereço de *i* o que também acontecerá com *t* quando da execução do comando t = p.

Se *p* é um ponteiro, a indicação **p* num programa acessa o conteúdo da variável para a qual *p* aponta. Assim podemos ter o seguinte programa:

```
#include <stdio.h>
main()
{
float *a, *b;
float Aux, x, y;
x = 1;          /* o conteúdo de x agora é igual a 1 */
y = 2;          /* o conteúdo de y agora é igual a 2 */
a = &x;         /* a aponta para x */
b = &y;         /* b aponta para y */
Aux = *a;       /* o conteúdo de Aux agora é 1 (conteúdo de x) */
*a = *b;        /* o conteúdo de x agora é 2 (conteúdo de y) */
*b = Aux;       /* o conteúdo de y agora é 1 */
printf("x = %f e y = %f \n", x, y);
}
```

Temos então um programa - certamente um pouco sofisticado - para permutar os conteúdos de duas variáveis.

Qualquer que seja o tipo de variável apontada por um ponteiro, pode-se "atribuir-lhe" a constante 0 (zero) e pode-se comparar um ponteiro com esta constante. O sistema oferece uma constante simbólica *NULL* que pode (e normalmente o é) ser utilizado no lugar do zero para, mnemonicamente, indicar mais claramente que este é um valor especial para um ponteiro. Como veremos daqui por diante, este valor especial de ponteiro será utilizado para inicializações de ponteiros, para *valores de escape* de estruturas de repetição e para retorno de funções quando alguma ação pretendida não é conseguida.

5.5 Passagem de parâmetros por referência no Turbo C 2.01

A utilização de ponteiros como parâmetros de funções permite a passagem de parâmetros por referência no Turbo C 2.01. Considere um parâmetro *p* do tipo ponteiro. Como *p* armazenará um endereço, se for passado para *p* o endereço de uma variável que possa ser apontada por ele, qualquer ação realizada no ponteiro afetará o conteúdo da variável.

O caso da função *Troca()*, comentada na seção anterior, poderia ser definida da seguinte forma:

```
void troca(float *a, float *b)
{
float Aux;
Aux = *a;
*a = *b;
*b = Aux;
}
```

e suas ativações deveriam ser feitas através de *Troca(*&x, &y*)*.

5.6 Uma urna eletrônica

A passagem de parâmetros *por referência* também é muito útil quando se pretende que uma função retorne mais de um valor. Um destes valores pode ser retornado pelo comando *return()* e os demais podem ser retornados para variáveis que foram passadas *por referência* para parâmetros da função.

O exemplo abaixo, uma melhor resposta de um exercício proposto no capítulo anterior, transforma um computador numa urna eletrônica para a eleição, em segundo turno, para a presidência de um certo país, às quais concorrem dois candidatos *Alibabá*, de número 89, e *Alcapone*, de número 93, sendo permitido ainda o voto em branco (número 99) e considerando como voto nulo qualquer voto diferente dos anteriores.

A função *Confirma()* deve retornar dois valores: o primeiro para, no caso de confirmação do voto, permitir sua contabilização e o segundo para, ainda no caso de confirmação do voto, interromper a estrutura *do while*, o que permitirá a recepção do voto seguinte. Observe também a passagem por referência do parâmetro da função *ComputaVoto()*. Há necessidade de que seja desta forma, pelo fato de que esta função alterará conteúdos de variáveis diferentes.

```
#include <stdio.h>
#include <ctype.h>
#include <dos.h>
#include <conio.h>

/*Função para confirmação do voto*/
int Confirma(char *s, char *p)
{
int r;
char Conf;
printf("Voce votou em %s! Confirma seu voto (SN)? ", s);
fflush(stdin);
```

```
scanf("%c", &Conf);
if (toupper(Conf) == 'S')
  {
  *p = 's';
  r = 1;
  }
else
  {
  *p = 'n';
        printf("\a Vote de novo: ");
  sound(1000);
  delay(80000);
  nosound();
  r = 0;
  }
return r;
}
/*Função para computar cada voto confirmado para o candidato*/
void ComputaVoto(int *p)
{
*p = *p + 1;
}
/*Função principal*/
main()
{
int Alibaba, Alcapone, Nulos, Brancos, Eleitores, Voto;
char Sim, Cont;
clrscr();
Alibaba = Alcapone = Nulos = Brancos = 0;
do
  {
  do
            {
            printf(" 89 - Alibaba \n 93 - Alcapone \n 99 - Branco \n");
            printf("Digite seu voto: ");
            scanf("%d", &Voto);
            switch (Voto)
            {
            case 89:
                     if (Confirma("Alibaba", &Sim) == 1)
                              ComputaVoto(&Alibaba);
                     break;
            case 93:
                     if (Confirma("Alcapone", &Sim) == 1)
                              ComputaVoto(&Alcapone);
                      break;
            case 99:
                     if (Confirma("Brancos", &Sim) == 1)
                              ComputaVoto(&Brancos);
                     break;
            default:
                     if (Confirma("Nulo", &Sim) == 1)
                              ComputaVoto(&Nulos);
                     break;
                     }
            clrscr();
```

```
                }
          while (Sim != 's');
          printf("Outro eleitor (S/N)? ");
          fflush(stdin);
          scanf("%c", &Cont);
               }
        while (toupper(Cont) == 'S');
        Eleitores = Alibaba + Alcapone + Brancos + Nulos;
        printf("Total de eleitores %d \n Alibaba %d \n Alcapone %d \n Brancos %d \n Nulos %d", Eleitores,
Alibaba, Alcapone, Brancos, Nulos);
        }
```

O arquivo de cabeçalhos *dos.h* contém as funções *sound(n)*, *nosound()* e *delay(n)*. A primeira emite um som de frequência *n* hertz; a segunda interrompe a emissão de som e a terceira suspende a execução do programa por *n* milissegundos.

A razão da chamada da função *fflush()* é a seguinte. Em alguns sistemas, quando algum dado de entrada é digitado para execução da função *scanf()*, os compiladores C não o armazena diretamente na posição de memória respectiva, armazenando-o inicialmente numa região chamada *buffer* para, ao final da execução da função de leitura transferir o conteúdo do *buffer* para a memória. Se quando da execução de uma função de leitura o conteúdo do *buffer* não estiver vazio, é este conteúdo (naturalmente, indesejado) que será armazenado na variável. A ativação de *fflush(stdin)* "descarrega" todo o *buffer* dos dados digitados no teclado e assim a função de leitura aguardará que o dado realmente pretendido seja digitado. É prudente, portanto, preceder leituras de caracteres e de cadeias de caracteres pela chamada de *fflush(stdin)*.

Observe que um ponteiro do tipo *char* é capaz de "armazenar" uma cadeia de caracteres (mais detalhes no capítulo 8). Observe também que utilizamos *inicializações sucessivas* no comando Alibaba = Alcapone = Nulos = Brancos = 0. Esta forma de inicializar variáveis também é válida, mas não será muito utilizada neste livro.

Para concluir (por enquanto) o estudo de ponteiros, vale ressalvar que, sendo variáveis capazes de receber endereços (portanto, valores do tipo *int*) pode-se somar e subtrair ponteiros. No capítulo 7, veremos um exemplo onde isto será útil.

5.7 Recursividade

Algumas funções matemáticas podem ser estabelecidas de tal forma que as suas definições utilizem, de modo recorrente, a própria função que se está definindo. Um exemplo trivial (no bom sentido) de um caso como este é a função *fatorial*. No ensino médio aprendemos que o fatorial de um número natural *n* é o produto de todos os números naturais de 1 até o referido *n*, ou seja, n! = 1 . 2 . 3 . ... . n. Como mostra o exemplo abaixo, é muito simples se escrever uma função (*função iterativa*) que calcule o fatorial de *n*: basta se inicializar um variável com 1 e, numa estrutura de repetição, calcular os produtos 1 x 2 = 2, 2 x 3 = 6; 6 x 4 = 24; 24 x 5 = 120; ...; etc., até multiplicar todos os naturais até *n*.

```
long int Fatorial(int n)
{
long int Fat;
int i;
Fat = 1;
for (i = 2; i <= n; i = i + 1)
  Fat = Fat * i;
return (Fat);
}
```

Embora o conceito anterior seja de simples compreensão, é matematicamente mais elegante definir o fatorial de um natural *n* por

$$n! = \begin{cases} 1, se\, n = 0\; ou\, n = 1 \\ n.(n-1)!, se\, n > 1 \end{cases}$$

Desta forma, o fatorial de *n* é definido a partir dos fatoriais dos naturais menores que ele. Isto significa que para o cálculo do fatorial de um determinado número natural há necessidade de que se *recorra* aos fatoriais dos naturais anteriores. Por exemplo, 4! = 4 . 3! = 4 . (3 . 2!) = (4 . 3) . (2 . 1!) = 4 . 3 . 2 . 1 = 24. Uma definição com estas características é dita uma definição por *recorrência* ou uma definição *recursiva*.

Um outro exemplo de uma definição recursiva foi dada no exercício 12 da seção 4.5: a *sequência de Fibbonaci* é a sequência (a_n) definida por

$$Fibb(n) = \begin{cases} 1, se\, n = 1\ ou\, n = 2 \\ Fibb(n-1) + Fibb(n-2), se\, n > 2 \end{cases}$$

Observe que o termo de ordem *n* é definido a partir de termos anteriores. Isto significa que para o cálculo de um determinado termo há necessidade de que se *recorra* a valores de todos os termos anteriores. Por exemplo, para a determinação de a_5 necessitamos conhecer a_4 e a_3; para a determinação destes dois, necessitamos conhecer a_2 e a_1.

Naturalmente, uma definição recursiva deve conter uma condição que interrompa a *recorrência*. Esta condição é chamada *condição de escape*. No caso do fatorial a condição de escape é n = 0 ou n = 1; na sequência de Fibbonaci, a condição de escape é n = 1 ou n = 2. A expressão que realiza propriamente a recorrência pode ser chamada *expressão de recorrência*

O surpreendente é que os ambientes para desenvolvimento de programas, de um modo geral, oferecem recursos para implementação de *funções recursivas* da mesma maneira que elas são escritas em matemática. Por exemplo, a implementação recursiva do fatorial em C pode ser feita simplesmente da seguinte forma:

```
long int FatRec(int n)
{
if ((n == 0) || (n == 1))
  return (1);
else
  return (n * FatRec(n - 1));
}
```

É interessante ter uma ideia do que acontece na recursividade. Quando se ativa uma função recursiva, cada nova chamada da mesma é empilhada na chamada *pilha de memória*, até que a condição de escape é atingida. A partir daí, cada ativação pendente é desempilhada (evidentemente, na ordem inversa do empilhamento) e as operações vão sendo realizadas.

Se ativarmos a função acima com n = 5 (com um comando printf("%d"%, FatRec(5)), por exemplo) teríamos a seguinte sequência de operações:

1 - Após a ativação de Fat(5)

Fat(5)	n	
5*Fat(4)	5	

2 - Após a ativação de Fat(4)

Fat(5)	n	Fat(4)	n	
5*Fat(4)	5	4*Fat(3)	3	

3 - Após a ativação de Fat(3)

Fat(5)	n	Fat(4)	n	Fat(3)	n	
5*Fat(4)	5	4*Fat(3)	3	3*Fat(2)	2	

4 - Após a ativação de Fat(2)

Fat(5)	n	Fat(4)	n	Fat(3)	n	Fat(2)	n	
5*Fat(4)	5	4*Fat(3)	3	3*Fat(2)	2	2*Fat(1)	1	

5 - Após a ativação de Fat(1)

Fat(5)	n	Fat(4)	n	Fat(3)	n	Fat(2)	n	
5*Fat(4)	5	4*Fat(3)	3	3*Fat(2)	2	2*1 = 2	1	

Fat(5)	n	Fat(4)	n	Fat(3)	n	
5*Fat(4)	5	4*Fat(3)	3	3*2 = 6	2	

Fat(5)	n	Fat(4)	n	
5*Fat(4)	5	4*6 = 24	3	

Fat(5)	n	
5*24 = 120	5	

Embora a utilização da recursividade apresente a vantagem de programas mais simples, ela tem o inconveniente de sacrificar a eficiência do programa. Isto ocorre devido à necessidade de chamadas sucessivas da função e das operações de empilhamento e desempilhamento, o que demanda um tempo maior de computação e uma maior necessidade de uso de memória. Esta observação faz com que a solução não recursiva (chamada, como já dissemos, *função iterativa*) seja preferível. No capítulo 7 apresentaremos um exemplo de uma função recursiva que é tão eficiente quanto a função iterativa.

Mesmo funções que não possuam intrinsecamente um definição recursiva pode ser implementada recursivamente, muitas das vezes com uma lógica mais fácil de compreender do que a da solução iterativa. Por exemplo, a função que determina o máximo divisor comum de dois inteiros dados apresentada na seção 5.1 pode ser escrita recursivamente da seguinte forma:

```
/*Função recursiva que retorna o máximo divisor comum de dois inteiros positivos dados*/
int MaxDivCom(int x, int y)
{
int Resto;
Resto = x % y;
if (Resto == 0)
	return y;
else
	return MaxDivCom(y, Resto);
}
```

Um outro exemplo interessante de recursividade é a implementação do jogo conhecido como *Torre de Hanói*, jogo que consiste em três torres chamadas *origem*, *destino* e *auxiliar* e um conjunto de n discos de diâmetros diferentes, colocados na torre *origem*, na ordem decrescente dos seus diâmetros. O objetivo do jogo é, movendo um único disco de cada vez e não podendo colocar um disco sobre outro de diâmetro menor, transportar todos os discos para pilha *destino*, podendo usar a torre *auxiliar* como passagem intermediária dos discos.

Indicando torre 1 → torre 2 o movimento do disco que no momento está parte superior da torre 1 para a torre 2, teríamos a seguinte solução para o caso n = 2:

1. origem → auxiliar
2. origem → destino
3. auxiliar → destino

Para n = 3, a solução seria:

1. origem → destino
2. origem → auxiliar
3. destino → auxiliar
4. origem → destino
5. auxiliar → origem
6. auxiliar → destino

7. origem → destino

Observe que os três movimentos iniciais transferem dois discos da torre *origem* para a torre *auxiliar*, utilizando a torre *destino* como auxiliar; o quarto movimento transfere o maior dos discos da *origem* para *destino* e os últimos movimentos transferem os dois discos que estão na *auxiliar* para *destino* utilizando *origem* como torre auxiliar.

Assim, a operação Move(3, origem, auxiliar, destino) - move três discos da *origem* para *destino* usando *auxiliar* como torre auxiliar - pode ser decomposta em três etapas:

1) Move(2, origem, destino, auxiliar) - move dois discos de origem para auxiliar usando destino como auxiliar;
2) Move um disco de origem para destino
3) Move(2, auxiliar, origem, destino) - move dois discos de auxiliar para destino usando origem como auxiliar.

O interessante é que é fácil mostrar que este raciocínio se generaliza para n discos, de modo que a operação Move(n, a, b, c) pode ser obtida com as seguintes operações:

1) Move(n-1, a, c, b)
2) Move um disco de a para c
3) Move(n-1, b, a, c)

O mais interessante ainda é que isto pode ser implementado em C, através do seguinte programa:

```
/* Programa que implementa o jogo Torre de Hanoi*/
#include <stdio.h>
void MoveDisco(char t1[10], char t2[10])
{
printf("%s --> %s \n", t1, t2);
}

void Hanoi(int x, char o[10], char a[10], char d[10])
{
if (x > 0)
  {
  Hanoi(x - 1, o, d, a);
  MoveDisco(o, d);
  Hanoi(x - 1, a, o, d);
  }
}

main()
{
int n;
printf("Digite o numero de discos \n");
scanf("%d", &n);
Hanoi(n, "origem", "auxiliar", "destino");
}
```

5.8 Usando funções de outros arquivos

Os compiladores C permitem que um programa utilize funções definidas em outros programas. Basta que o referido programa seja incluído na instrução *#include "NomeArquivo"*. Por exemplo, imagine que a declaração de variáveis e a função abaixo

```
#include <stdio.h>
int x, y;
int MaxDivCom(int a, int b)
{
int Resto;
```

```
Resto = a % b;
while (Resto != 0)
  {
  a = b;
  b = Resto;
  Resto = a % b;
  }
return b;
}
```

estejam num arquivo *mdc.c*. Como a matemática prova que o produto de dois números inteiros é igual ao produto do máximo divisor comum dos números pelo mínimo múltiplo comum, a função *MaxDivCom()* poderia ser utilizada para se escrever um programa para o cálculo do mínimo múltiplo comum de dois números dados. Este programa poderia utilizar as variáveis x, y e *Mdc* declaradas no arquivo *mdc.c*. Teríamos então o seguinte programa:

```
/*Programa que calcula o minimo multiplo comum de dois numeros utilizando uma função definida em outro arquivo*/
#include <stdio.h>
#include "mdc.c"
main()
{
int m, Mmc;
printf("Digite os numeros ");
scanf("%d %d", &x, &y);
m = MaxDivCom(x, y);
Mmc = (x * y)/m;
printf("Mmc(%d, %d) = %d \n", x, y, Mmc);
}
```

A referência ao arquivo que vai ser “incluído” no programa pode ser feita com o *caminho* do arquivo escrito entre aspas ou com o nome do arquivo entre < >, se ele está gravado na pasta padrão dos arquivos de cabeçalho *.h*.

5.9 "Tipos" de variáveis

Variáveis locais

Como foi dito na seção 5.1, as variáveis declaradas no interior de uma função (*variáveis locais*, para lembrar) só são acessáveis por instruções desta função. Na realidade, elas só existem durante a execução da função: são "criadas" quando a função é ativada e são "destruídas" quando termina a execução da função. Por esta última razão, variáveis locais também são chamadas *variáveis automáticas* e *variáveis dinâmicas*. Também são variáveis locais os parâmetros da função. Isto explica a necessidade de declará-los, definindo-se seus identificadores e seus tipos de dados.

Variáveis globais e o modificador *extern*

Se uma variável deve ser acessada por mais de uma função ela deve ser declarada fora de qualquer função, sendo chamada, neste caso, de *variável global*. Uma variável global pode ser referenciada em qualquer função do programa e, embora isto não seja aconselhável, pode-se identificar uma variável local com o mesmo identificador de uma variável global. Neste caso, referências ao identificador comum dentro da função na qual a variável local foi definida refere-se a esta variável local. Isto, naturalmente, impede a função de acessar a variável global.

O *modificador de variável extern* permite que um programa utilize variáveis definidas e inicializadas em funções de um outro arquivo que tenha sido "incluído" através da instrução #include "NomeArquivo", como visto na seção 5.8. No capítulo seguinte apresentaremos um exemplo bastante esclarecedor do uso do

modificador *extern*.

Variáveis estáticas

Como uma variável local deixa de existir quando se encerra a execução da função, o último valor nela armazenado é perdido. Pode ocorrer que o último valor armazenado numa variável local seja necessário para uma chamada subsequente da função. Após a execução de uma função, o último valor armazenado numa variável local pode ser preservado para ser utilizado numa chamada posterior da função através do *modificador de variável static*. No Turbo C 2.01 e no Turbo C++ 3.0, uma variável local *static* deve ser inicializada por um valor constante ou o endereço de uma variável global, quando da sua declaração. Neste caso, considerando que receberá endereços, uma variável *static* deve ser definida como um ponteiro.

Por exemplo, o programa abaixo utiliza uma variável estática para guardar o valor de *Termo* em cada ativação da função *GeraPA()* para, com este valor, obter o valor do termo seguinte.

```
#include <stdio.h>
#include <conio.h>
int a1;
int GeraPA(int r)
{
static int *Termo = &a1;
*Termo = *Termo + r;
return (*Termo);
}
main()
{
int i, Razao;
clrscr();
printf("Digite o primeiro termo e a razÆo: ");
scanf("%d %d", &a1, &Razao);
printf("Progressao Aritmetica de primeiro termo %d e razao %d: \n%d ", a1, Razao, a1);
for (i = 1; i <= 9; i = i + 1)
  printf("%d ", GeraPA(Razao));
}
```

Naturalmente, concordo com o leitor que está pensando que existem programas para geração de progressões aritméticas bem mais simples.

Uma outra aplicação de variáveis *estáticas* pode ser obtida numa função que determine a decomposição em fatores primos de um inteiro dado, exercício proposto no capítulo anterior. Um algoritmo para determinar a decomposição em fatores primos de um inteiro é efetuar divisões sucessivas pelos primos divisores, sendo o número de divisões pelo primo realizadas a sua multiplicidade. Por exemplo, para se obter a decomposição de 1.400 teríamos

1400	2
700	2
350	2
175	5
35	5
7	7
1	$1400 = 2^3 x 5^2 x 7$

As funções abaixo, a primeira iterativa e a segunda recursiva, retornam a decomposição de um inteiro passado para o parâmetro *n*.

```
void DecompFatores(int n)
{
int d, m;
d = 2;
```

```
while (n > 1)
  {
  m = 0;
  while (n % d == 0)
           {
           m++;
           n = n / d;
           }
        if (m > 0)
           printf("\t%d\t%d\n", d, m);
        d++;
  }
}
void DecompFatoresRec(int n)
{
static int d = 2;
int m = 0;
if (n > 1)
  {
  while (n % d == 0)
           {
           m++;
           n = n / d;
           }
  if (m > 0)
           printf("\t%d\t%d\n", d, m);
  d++;
  DecompFatoresRec(n);
  }
}
```

Observe que como *d* foi definida como estática, o seu valor da ultima execução da função, obtido através do comando *d++*, é preservado permitindo que um novo divisor primo seja procurado.

O valor de uma variável estática é preservado para uma próxima execução da função pelo fato de que, na verdade, ela não é destruída quando se encerra a execução da função, como se ela fosse uma variável global. A diferença entre variável local estática e variável global é que a primeira continua só podendo ser acessada pela função onde ela foi definida.

5.10 Uma aplicação à História da Matemática

Em 1817, Christian Goldbach, um professor de Matemática e de História nascido na Alemanha, numa carta que escreveu ao matemático Leonard Euler fez a seguinte afirmação: todo número par maior quer dois é a soma de dois números primos. Embora a veracidade desta afirmação, conhecida hoje como *Conjectura de Goldbach*, já tenha sido verificada para todos os números pares menores do que 12×10^{17} (http://www.ieeta.pt/~tos/goldbach.html, acessada em 14/12/2008), ainda não se conseguiu prová-la integralmente, apesar dos esforços já despendidos por muitos matemáticos.

O programa abaixo, permite a verificação da *Conjectura de GoldBach* para qualquer número par menor que 32.767.

```
#include <math.h>
#include <stdio.h>
int Primo(int n)
{
int d = 2;
float r = sqrt(n);
```

```
while (n%d != 0 && d <= r)
  d++;
if (d <= r)
  return 0;
else
  return 1;
}
void Goldbach(int n, int &a, int &b)
{
int i = 2;
while (!Primo(i) || !Primo(n - i))
  i++;
a = i;
b = n - i;
}
main()
{
int x, y, z;
printf("Digite um inteiro par ");
scanf("%d", &x);
Goldbach(x, y, z);
printf("Primos cuja soma , igual a %d:  %d e %d \n", x, y, z);
}
```

5.11 Exercícios propostos

1. Escreva uma função que retorne o k-ésimo dígito (da direita para esquerda) de um inteiro n, k e n dados. Por exemplo, K_esimoDigito(2845, 3) = 8.

2. O *fatorial ímpar* de um número n ímpar positivo é o produto de todos os números ímpares positivos menores do que ou iguais a n. Indicando o *fatorial ímpar* de n por n| temos, n| = 1 . 3. 5 . ... . n. Por exemplo, 7| = 1 . 3 . 5 . 7 = 105. Escreva funções iterativas e recursivas para a determinação do fatorial ímpar de um inteiro ímpar dado.

3. Como na questão anterior, o *fatorial primo (ou primorial*) de um número primo positivo é o produto de todos os primos positivos menores do que ou iguais a ele: p# = 2 . 3 . 5 . 7 . ... .p (sendo 2# = 2). Por exemplo, 7# = 2 . 3 . 5 . 7 = 210. Escreva um programa que determine o fatorial primo de um primo dado.

4. Escreva funções, iterativa e recursiva, que retornem a soma dos algarismos de um inteiro positivo dado.

5. Escreva uma função recursiva que retorne o n-ésimo termo da sequência de Fibbonaci, n dado.

6. Escreva uma função recursiva que gere uma tabuada de multiplicação, exibindo-a no formato (para posicionar a saída pode-se utilizar a função *gotoxy()*).

```
1x2 = 2          1x3 = 3          1x4 = 4          . . .
2x2 = 4          2x3 = 6          2x4 = 8          . . .
3x2 = 6          3x3 = 9          3x4 = 12         . . .
. . .            . . .            . . .            . . .
9x2 = 18         9x3 = 27         9x4 = 36         . . .
```

7. Escreva uma função recursiva que determine o *mínimo múltiplo comum* de dois inteiros dados.

8. Escreva funções, recursiva e iterativa, que implementem a função *pow()*.

Observação

Propostas de soluções dos exercícios propostos podem ser solicitadas através de mensagem eletrônica para jaime@ccen.ufal.br com assunto RESPOSTAS LIVRO C, anexando o formulário abaixo devidamente preenchido.

Nome	Categoria[1]	Instituição[2]	Curso[2]	Cidade/Estado

[1]Categoria: docente, estudante, autodidata
[2]Se docente ou estudante

6 Vetores

6.1 O que são vetores

No exemplo 6 da seção 4.6 discutimos uma função para a determinação da média de uma relação de números dados. Para tal, utilizamos uma variável simples para receber os números, sendo que cada vez que um número, a partir do segundo, era recebido o anterior era "perdido". Ou seja, a relação de números não era armazenada. Imagine que a relação fosse uma relação de notas escolares e além da média se quisesse também saber a quantidade de alunos que obtiveram nota acima da média ou uma outra medida estatística (*desvio médio*, por exemplo) que dependesse da média. Neste caso, haveria a necessidade de que a relação fosse redigitada o que, além da duplicidade do trabalho, facilitaria os erros de digitação. É importante então que exista uma "variável" capaz de armazenar vários valores simultaneamente de tal forma que se possa acessar cada um deles independentemente de se acessar os demais.

Um outro exemplo é o caso do exemplo 2 da seção 4.6. Lá queríamos a relação dos divisores de um inteiro dado e estes divisores eram apenas exibidos, não sendo armazenados, como recomendado na seção 2.9. Até aquele momento, a dificuldade de se armazenar os divisores residia no fato de que não se sabe *a priori* o número de divisores de um inteiro dado e, portanto, não saberíamos quantas variáveis deveríamos declarar.

Um *vetor* é um conjunto de variáveis de um mesmo tipo de dado as quais são acessadas e referenciadas através da aposição de *índices* ao identificador do vetor.

6.2 Declaração de um *vetor unidimensional*

Um *vetor unidimensional* (ou simplesmente *vetor*) é declarado através da seguinte sintaxe:

Tipo de dado Identificador[n];

onde *Tipo de dado* fixará o tipo de dado das variáveis *componentes* do vetor e *n* indicará o número das tais componentes.

Por exemplo, a declaração

```
int Vetor[10]
```

definirá um conjunto de dez variáveis do tipo *int*, enquanto que a declaração

```
char Cadeia[100]
```

definirá um conjunto de cem variáveis do tipo *char*. Como cada variável do tipo *int* utiliza dois bytes de memória e cada variável do tipo *char* utiliza apenas um byte, a variável *Vetor* ocupará vinte bytes de memória enquanto que a variável *Cadeia* ocupará cem bytes.

Os compiladores C possuem uma função de biblioteca, *sizeof()*, que retorna o número de bytes ocupado por uma variável ou por um vetor. Por exemplo, o programa

```
#include <stdio.h>
main()
{
float x;
int v[30];
printf("Numero de bytes de x = %d \nNumero de bytes de v = %d \n", sizeof(x), sizeof(v));
}
```

exibirá na tela a seguinte saída:

```
Numero de bytes de x = 4
Numero de bytes de v = 60
```

Cada componente de um vetor pode ser acessada e referenciada através de índices associados ao identificador do vetor, sendo o índice da primeira componente igual a zero. Assim, as componentes do vetor *Cadeia* do exemplo acima serão identificadas por Cadeia[0], Cadeia[1], ..., Cadeia[99].

O índice de uma componente pode ser referido através de uma expressão que resulte num valor

inteiro. Por exemplo, a sequência de comandos

```
int i, Quadrados[100];
for (i = 1; i <= 100; i = i + 1)
  Quadrados[i - 1] = i * i;
```

armazena no vetor *Quadrados* os quadrados dos cem primeiros inteiros positivos.

Uma coisa em que o programador em C deve se preocupar é o fato de que os compiladores C não verificam se os valores atribuídos a um índice estão dentro dos limites definidos na declaração do vetor. Se os limites não forem obedecidos, podem ocorrer erros de lógica na execução programa ou conflitos com o sistema operacional (provocando, até mesmo, *travamento* no sistema).

6.3 Vetores e ponteiros

É muito importante observar que o identificador de um vetor em C é, na verdade, um ponteiro que aponta para o primeiro elemento do vetor. Quando se declara

```
int v[10];
```

está se reservando um conjunto de dez posições de memória contíguas, cada uma delas com dois bytes, como mostra a figura abaixo:

A partir daí, qualquer referência ao identificador *v* é uma referência ao endereço da componente v[0], de tal forma que, se tivermos a declaração

```
int *p;
```

os comandos

```
p = &v[0];          p = v;
```

executam a mesma ação: armazenam no ponteiro *p* o endereço de v[0].

Sendo um ponteiro que aponta para sua primeira componente, um vetor pode ser um parâmetro de uma função, podendo receber endereços. Dessa forma, qualquer ação realizada no vetor afetará o conteúdo do vetor passado como argumento; ou seja, a passagem de parâmetros do tipo vetor é sempre feita *por referência*. Esta observação também justifica a possibilidade de que parâmetros do tipo vetor sejam declarados como um ponteiro: void funcao(int *v).

6.4 Lendo e escrevendo um vetor

Como foi dito na seção 6.1, vetores servem para armazenar uma relação de dados do mesmo tipo. Uma função para fazer este armazenamento depende do conhecimento ou não da quantidade de elementos da relação. Na hipótese do número de elementos da relação ser conhecido, basta usar uma função, do tipo *void*, com dois parâmetros: um para receber o vetor que vai armazenar a relação e outro para receber a quantidade de elementos da relação. Dentro da função, pode-se utilizar um comando *for*.

```
#include <stdio.h>
void ArmazenaRelacaoN(int *v, int t)
{
int i;
printf("Digite os elementos da relacao \n");
for (i = 0; i < t; i++)
  scanf("%d", &v[i]);
}
```

Se o número de elementos da relação não é conhecido *a priori*, deve-se utilizar um *flag* para encerrar a entrada dos dados, de acordo com o que foi comentado na seção 4.6. É importante que a função, para utilizações posteriores, determine o número de elementos da relação. Este valor pode ser retornado através

do comando *return()*. Dentro da função pode-se utilizar um comando *while* que deverá ser executado enquanto o dado de entrada for diferente do *flag* escolhido.

```
int ArmazenaRelacao(int *v)
{
int i;
i = 0;
printf("Digite os elementos da relacao (-1 para encerrar)");
scanf("%d", &v[i]);
while (v[i] != -1)
  {
  i = i + 1;
  scanf("%d", &v[i]);
  }
return (i);
}
```

Observe a dupla finalidade da variável *i*: ela serve para indexar as componentes e para determinar a quantidade de elementos da relação.

Para se exibir uma relação de dados armazenados num vetor, basta uma função com dois parâmetros: um para receber o vetor onde a relação está armazenada e o outro para receber a quantidade de elementos da relação. Esta quantidade está armazenada em alguma variável: ela foi um dado de entrada ou foi determinada na ocasião do armazenamento da relação.

```
void ExibeRelacao(int *v, int t)
{
int i;
for (i = 0; i < t; i++)
  printf("%d ", v[i]);
printf("\n");
}
```

6.5 Exemplos Parte IV

Os exemplos a seguir, além de reforçar vários aspectos da utilização de vetores, são muito úteis no desenvolvimento da lógica de programação.

1. Para introduzir o estudo dos vetores nos referimos, na seção 6.1, ao problema de se determinar o número de alunos de uma turma que obtiveram notas maiores que a média. Com a utilização de vetores, podemos calcular a tal média e depois "percorrer" novamente o vetor, comparando cada nota com a referida média. Teríamos então o seguinte programa:

```
#include <stdio.h>
int ArmazenaNotas(float *v)
{
int i;
i = 0;
printf("Digite as notas (-1 para encerrar)");
scanf("%f", &v[i]);
while (v[i] != -1)
  {
  i = i + 1;
  scanf("%f", &v[i]);
  }
return (i);
}

/*Função para determinar a média de uma relação de números*/
float Media(float *v, int t)
{
```

```
int i;
float Soma;
Soma = 0;
for (i = 0; i < t; i++)
  Soma = Soma + v[i];
return (Soma/t);
}

main()
{
int i, Quant, NotasBoas;
float *p, Med;
Quant = ArmazenaNotas(p);
Med = Media(p, Quant);
NotasBoas = 0;
for (i = 0; i < Quant; i = i + 1)
  if (p[i] > Med)
        NotasBoas = NotasBoas + 1;
printf("Media das notas: %f \n", Med);
printf("Notas maiores que a media: %d", NotasBoas);
}
```

2. Imagine que quiséssemos o *desvio médio* das notas do exemplo anterior. Esta medida estatística é assim definida: o *desvio de um elemento em relação à média aritmética* é a diferença entre o elemento e a média aritmética da relação; o *desvio médio* é a média aritmética dos valores absolutos dos desvios.

Podemos então, como fizemos acima, escrever uma função para calcular a média da relação de notas, armazenar os valores absolutos dos desvios em um vetor e utilizar a função que calcula a média para calcular a média destes valores absolutos.

```
float DesvioMedio(float *v, int t)
{
int i;
float  *d;
float Med;
Med = Media(v, t);
for (i = 0; i < t; i++)
  d[i] = abs(v[i] - Med);
return(Media(d, t));
}
```

Observe que este exemplo ilustra outra vantagem do uso de funções: a mesma função *Media* foi utilizada para determinação de médias de relações diferentes.

3. Imaginemos agora que queiramos uma função que retorne o maior valor de uma relação armazenada em um vetor. Uma possível solução é supor que o maior valor procurado é o primeiro elemento do vetor e, em seguida, percorrer o vetor comparando cada componente com o valor que, até o momento, é o maior, substituindo o valor deste maior elemento quando se encontra uma componente maior que ele.

```
float MaiorElemento(float *v, int t)
{
int i;
float Maior;
Maior = v[0];
for (i = 1; i < t; i = i + 1)
  if (v[i] > Maior)
        Maior = v[i];
return(Maior);
}
```

Observe que a função acima retorna o maior elemento armazenado no vetor, mas não retorna a posição deste maior elemento, o que em muitas situações é indispensável. Para isto é necessário um parâmetro com passagem por referência, como visto no capítulo anterior. Este parâmetro recebe o valor zero e em seguida o

valor da posição onde foi encontrada uma componente maior do que *Maior*.

```
float MaiorElemento(float *v, int t, int *p)
{
int i;
float Maior;
Maior = v[0];
*p = 0;
for (i = 1; i < t; i = i + 1)
  if (v[i] > Maior)
        {
        Maior = v[i];
        *p = i;
        }
return(Maior);
}
```

Uma chamada desta função vai requerer, além do vetor *p* onde está armazenada a relação, duas variáveis, digamos *Maior* e *Pos*; nestas condições a ativação da função será feita através do comando

```
Maior = MaiorElemento(p, Quant, &Pos);
```

onde *p* é o vetor onde está armazenada a relação, *Quant* é a quantidade de elementos da relação e *&Pos* é o endereço da variável que irá armazenar a posição da ocorrência do maior elemento procurado.

Vale ressaltar que, na função acima, adotamos a passagem por referência "formato Turbo C 2.01" que também funciona no Turbo C++ 3.0.

4. O exemplo a seguir tem o objetivo de mostrar que o índice de acesso às componentes de um vetor pode ser dado através de expressões, como já foi dito anteriormente. O exemplo mostra uma função que, recebendo dois vetores com a mesma quantidade de elementos, gera um vetor intercalando as componentes dos vetores dados. Assim se v_1 = {4, 8, 1, 9} e v_2 = {2, 5, 7, 3} a função deve gerar o vetor v = {4, 2, 8, 5, 1, 7, 9, 3}. Observe que as componentes ímpares de *v* são os elementos de v_1 e as componentes pares são os elementos de v_2.

```
void IntercalaVetor(float *v1, float *v2, float *v, int t)
{
int i;
for (i = 0; i < 2 * t; i = i + 1)
  if (i % 2 == 1)
        v[i] = v2[(i - 1)/2];
  else
        v[i] = v1[i/2];
}
```

5. Agora apresentaremos um exemplo que mostra um vetor cujas componentes são cadeias de caracteres, além de mostrar como um vetor de caracteres pode ser inicializado. Trata-se de uma função que retorna o nome do mês correspondente a um número dado, que poderia ser usada num programa que escrevesse por extenso uma data da no formato dd/mm/aaaa.

```
char *NomeMes(int n)
{
char *Nome[13] = {"Mes ilegal", "Janeiro", "Fevereiro", "Marco", "Abril", "Maio", "Junho", "Julho",
"Agosto", "Setembro", "Outubro", "Novembro", "Dezembro"};
if ((n > 0) && (n < 13))
  return(Nome[n]);
else
  return(Nome[0]);
}
```

Observe que a inicialização do vetor se faz quando da sua declaração, com a enumeração dos valores das componentes entre chaves, como a matemática faz com conjuntos. Observe também que a função *NomeMes()* retorna um ponteiro para uma variável do tipo *char*, o que significa que retornará uma cadeia de caracteres (uma *string*), pois, como veremos no próximo capítulo, uma *string* é um vetor de caracteres. Este

fato também é indicado no vetor *Nome* que é um vetor de ponteiros para variáveis do tipo *char*.

6. Este exemplo mostra uma situação em que há a necessidade de vários vetores. Trata-se de um programa para administrar os pedidos de uma lanchonete, cujo cardápio é o seguinte:

Codigo	Produto	Preço
101	Refrigerante	1.20
102	Suco	1.00
103	Sanduíche	2.50
104	Salgado	1.00
105	Torta	2.00

Uma possível solução é considerar três vetores globais, inicializados como no exemplo anterior, de tal forma que seja possível para cada pedido gerar vetores com os códigos dos produtos solicitados e, a partir daí, se possa armazenar as quantidades de cada item do pedido.

```
#include <stdio.h>
#include <conio.h>
int Cod[5] = {101, 102, 103, 104, 105};
char  *Prod[5] = {"Refrigerante", "Suco", "Sanduiche", "Salgado", "Torta"};
float Precos[5] = {1.20, 1.00, 2.50, 1.00, 2.00};

int Pedido(int *c, int *q, float *v)
{
int i = 0;
do
  {
  do
            {
               puts("Código (100 para encerrar o pedido):");
            scanf("%d", &c[i]);
               if (c[i] < 100 || c[i] > 105)
                     printf("\a Codigo invalido");
            }
  while (c[i] < 100 || c[i] > 105);
  if (c[i] != 100)
            {
            puts("Quantidade");
            scanf("%d", &q[i]);
            v[i] = q[i]*Precos[c[i] - 101];
            i++;
            }
  }
while (c[i] != 100);
return i;
}

void ExibePedido(int *c, int *q, float *v, int t)
{
int i;
float Total = 0.0;
clrscr();
puts("Código      Discriminação  Quantidade       Valor \n");
for (i = 0; i < t; i++)
  {
  printf("%d  %s  %10.d %10.2f\n", Cod[c[i] - 101], Prod[c[i] - 101], q[i], v[i]);
  Total = Total + v[i];
  }
printf("\nValor total do pedido: %.2f\n\n", Total);
}

main()
{
```

```
int NumItens, *Itens, *Quant;
float *Valor;
char S = 'n';
do
  {
  clrscr();
  NumItens = Pedido(Itens, Quant, Valor);
  if (NumItens != 0)
          ExibePedido(Itens, Quant, Valor, NumItens);
  puts("Outro pedido (S/N)?");
  fflush(stdin);
  scanf("%c", &S);
  }
while (toupper(S)  != 'N');
}
```

7. Mostraremos agora um exemplo cujo objetivo é desenvolver o raciocínio recursivo utilizando vetores. Trata-se de uma função recursiva para a determinação da soma das componentes de um vetor, coisa já feita iterativamente no interior da função *Media()* desenvolvida no exemplo 1. Ora, como uma função que trata vetores recebe sempre o vetor e o seu tamanho, podemos recursivamente "diminuir" o tamanho do vetor através do decremento do parâmetro respectivo até que o vetor "tenha apenas uma componente", quando então a soma das componentes se reduzirá a esta "única" componente.

```
/*Função recursiva que retorna a soma das componentes de um vetor*/
int SomaCompRec(float *v, int t)
{
if (t == 1)
  return v[i – 1];
else
  return v[t - 1] + SomaComRec(v, t – 1);
}
```

6.6 Vetores multidimensionais

Na seção 6.1 foi dito que um vetor é um conjunto de variáveis de mesmo tipo, chamadas componentes do vetor. A linguagem C permite que as componentes de um vetor sejam também vetores, admitindo que se armazene uma *matriz* da matemática, uma *tabela de dupla entrada* que, por exemplo, enumere as distâncias entre as capitais brasileiras ou um livro considerando a página, a linha e a coluna em que cada caractere se localiza.

A declaração de um vetor multidimensional é uma extensão natural da declaração de um vetor unidimensional:

Tipo de dado Identificador[n_1][n_2] ... [n_k];

onde k indica a *dimensão* e n_1, n_2, ..., n_k indicam o número de componentes em cada *dimensão*.

Por exemplo, a declaração

```
int Mat[10][8];
```

define um vetor de dez componentes, cada uma delas sendo um vetor de oito componentes. Ou seja, *Mat* é um conjunto de 10 x 8 = 80 variáveis do tipo *int*.

Para um outro exemplo, a declaração

```
char Livro[72][30][30];
```

define uma variável capaz de armazenar os caracteres de um livro com até setenta duas páginas, cada página possuindo trinta linhas e cada linha possuindo trinta colunas.

A inicialização de um vetor multidimensional também segue o padrão da inicialização de um vetor unidimensional, com a ressalva de que as componentes, que agora são vetores, devem estar entre chaves. Por exemplo, se quiséssemos um vetor bidimensional para armazenar os números de dias dos meses do ano, fazendo a distinção entre anos bissextos e não bissextos poderíamos declarar e inicializar um vetor

DiasMeses da seguinte forma:

int DiasMeses = {{0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}, {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}};

onde a primeira componente refere-se aos dias dos meses de um ano não bissexto e a segundo, aos dias dos meses de um ano bissexto (este vetor será utilizado num exemplo a seguir e, na ocasião, será explicada a razão da primeira componente de cada vetor componente ser zero).

Um vetor bidimensional é usualmente chamado de *matriz* e os números de componentes são chamados, respectivamente, *número de linhas* e *número de colunas*. Estes dois números separados por x (que é lido *por*) é a *ordem* da matriz. Assim, a variável *Mat* do exemplo acima está apta a armazenar até uma matriz de ordem 8 x 10. Estas denominações, emprestadas da matemática, são justificadas pelo fato de que, embora as componentes de um vetor bidimensional sejam armazenadas de forma consecutiva (a primeira componente do segundo vetor logo após a última componente do primeiro vetor), uma matriz, para facilitar sua compreensão, pode ser imaginada como constituída de *linhas* e de *colunas*. Por exemplo, o vetor *DiasMeses* do exemplo acima pode ser imaginado como sendo

0	31	28	31	30	31	30	31	31	30	31	30	31
0	31	29	31	30	31	30	31	31	30	31	30	31

facilitando a compreensão de referências do tipo *DiasMeses[1][2]* que indica o elemento da segunda linha (a primeira é de índice zero) e da terceira coluna (a primeira é de índice zero).

Se o número de linhas e o número de colunas de uma tabela são conhecidos, o seu armazenamento em uma matriz é muito simples. Basta utilizar dois comandos *for* aninhados, controlados pelo número de linhas e pelo número de colunas, respectivamente.

```
void ArmazenaTabelaMN(float Mat[10][10], int m, int n)
{
int i, j;
printf("Digite, por linha, os elementos da matriz");
for (i = 0; i < m; i = i + 1)
  for (j = 0; j < n; j = j + 1)
        scanf("%f", &Mat[i][j]);
}
```

Se o número de linhas e o número de colunas de uma tabela não são conhecidos, pode-se usar um duplo *while* aninhado, definindo-se um *flag* para encerramento da digitação dos elementos de cada linha e um outro *flag* para encerramento da digitação da matriz. Naturalmente, a função deverá retornar o número de linhas e o número de colunas da tabela, o que justifica os ponteiros *m* e *n* da proposta abaixo.

```
void ArmazenaTabela(float Mat[10][10], int &m, int &n)
{
int i, j;
printf("Digite, por linha, os elementos da matriz (-1 para encerrar cada linha e -2 para encerrar a matriz");
i = 0;
j = 0;
scanf("%f", &Mat[i][j]);
while (Mat[i][j] != -2)
  {
  while (Mat[i][j] != -1)
        {
        j = j + 1;
        scanf("%f", &Mat[i][j]);
        n = j;
        }
  i = i + 1;
  j = 0;
  scanf("%f", &Mat[i][j]);
  }
m = i;
}
```

Uma função para exibir uma tabela armazenada numa matriz também é muito simples. Basta, para que a matriz seja exibida na forma de tabela, mudar a linha cada vez que a exibição de uma linha é concluída.

```
void ExibeTabela(float Mat[10][10], int m, int n)
{
int i, j;
for (i = 0; i < m; i = i + 1)
  {
  for (j = 0; j < n; j = j + 1)
            printf("%.1f   ", Mat[i][j]);
  printf("\n");
  }
}
```

6.7 Exemplos Parte V

1. Um dos exemplos do capítulo 3 apresentava uma função que recebia uma data e fornecia o dia da semana correspondente. Neste programa precisamos calcular o número de dias do ano decorridos até aquela data. Com a utilização da matriz *DiasMeses* comentada acima, podemos escrever facilmente uma função com este objetivo.

```
int DiaAno(int d, int m, int a)
{
int DiasMes[2][13] = {{0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
                      {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31}};
int i;
if (((a % 4 == 0) && (a % 100 != 0)) || (a % 400 == 0))
  for (i = 1; i < m; i = i + 1)
            d = d + DiasMes[1][i];
else
  for (i = 1; i < m; i = i + 1)
            d = d + DiasMes[0][i];
return(d);
}
```

Lembrando que o valor de uma expressão lógica é 1 (um) se ela for verdadeira e 0 (zero) se ela for falsa, poderíamos armazenar o valor da expressão lógica que garante que um ano é bissexto e utilizar o conteúdo desta variável para escolher a componente do vetor *DiasMeses* que será utilizada: 1 (um) se for bissexto e (0) zero, caso contrário.

```
int DiaAno(int d, int m, int a)
{
int i,Biss;
Biss =  ((a % 4 == 0) && (a % 100 != 0)) || (a % 400 == 0);
for (i = 1; i < m; i = i + 1)
  d = d + DiasMes[Biss][i];
return(d);
}
```

A razão de termos considerado a primeira componente igual a 0 (zero) foi para compatibilizar o número correspondente a cada mês com a componente do vetor, já que (repetindo pela última vez) a primeira componente de um vetor é de índice zero.

2. Como no exemplo em que o próprio sistema gerou os quadrados dos cem primeiros números inteiros, o sistema pode gerar uma matriz. Para exemplificar isto, apresentaremos uma função que gera a *matriz identidade de ordem n*. Para um inteiro positivo dado, a *matriz identidade de ordem n* é a matriz $I_n = (i_{rs})$, de ordem nxn, dada por $i_{rs} = 1$, se r = s, e $i_{rs} = 0$, se $r \neq s$. Esta matriz é muito importante no estudo das matrizes, sendo utilizada, por exemplo, para a determinação da *matriz inversa* de uma matriz inversível. Por exemplo, se n = 3, temos

$$I_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

```
void GeraMatrizUnidade(int Mat[10][10], int m)
{
int i, j;
for (i = 0; i < m; i = i + 1)
    for (j = 0; j < m; j = j + 1)
        if (i == j)
          Mat[i][j] = 1;
        else
          Mat[i][j] = 0;
}
```

3. Quando o número de linhas de uma matriz é igual ao número de colunas a matriz é dita *matriz quadrada*. Neste caso, os elementos de índices iguais constituem a *diagonal principal*. A soma dos elementos da diagonal principal de uma matriz quadrada é o *traço* da matriz. Como mais um exemplo de programas que manipulam matrizes, a função abaixo determina o *traço* de uma matriz quadrada dada. Observe que para percorrer a diagonal principal não há necessidade de um duplo *for*.

```
float Traco(float Mat[10][10], int m, int n)
{
int i;
float Tr;
Tr = 0;
if (m == n)
        {
        for (i = 0; i < m; i = i + 1)
            Tr = Tr + Mat[i][i];
        return(Tr);
        }
else
        printf("A matriz nao e quadrada");
}
```

4. Uma tabela que enumere as distâncias entre várias cidades é uma matriz *simétrica*: os termos simétricos em relação à *diagonal principal* são iguais, ou seja Mat[i][j] = Mat[j][i]. Obviamente, a digitação de uma matriz com esta propriedade pode ser simplificada, devendo-se digitar apenas os termos que estão acima da diagonal principal.

```
void ArmazenaMatrizSimetrica(float Mat[10][10], int m)
{
int i, j;
printf("Digite, por linha, os elementos da matriz, a partir da diagonal");
for (i = 0; i < m; i = i + 1)
        for (j = i; j < m; j = j + 1)
          {
          scanf("%f", &Mat[i][j]);
          Mat[j][i] = Mat[i][j];
          }
}
```

Observe que a inicialização de *j* no segundo comando *for* foi com o valor de cada *i* do primeiro. A razão disto é que só serão digitados os termos acima da diagonal principal, termos em que $j \geq i$.

5. Nos exemplos anteriores, sempre "percorremos a matriz pelos elementos de suas linhas". O próximo exemplo mostra um caso em que é necessário percorrer as colunas. Trata-se de uma questão muito comum da totalização das colunas de uma tabela.

```
void TotalizaColunas(float Mat[10][10], int m, int n)
```

```
{
int i, j;
for (j = 0; j < n; j = j + 1)
  {
  Mat[m][j] = 0;
  for (i = 0; i < m; i = i + 1)
           Mat[m][j] = Mat[m][j] + Mat[i][j];
      }
}
```

6.8 Uma aplicação esportiva

Nesta seção, apresentaremos um programa para administrar o placar de um set de um jogo de vôlei de praia. De acordo com as regras em vigoravam nas Olimpíadas de Pequim (2008), para uma equipe vencer um set de uma partida ela deveria obter um mínimo de 21 pontos para os sets “normais” ou de 15 pontos para um set de desempate, desde que a diferença entre sua pontuação e a do adversário fosse superior ou igual a dois.

```
#include <stdio.h>
#include <conio.h>
#include <math.h>

void MostraPlacar(char *Time1, char *Time2, int Pontos1, int Pontos2)
{
printf("%20s %2d x %2d %-20s\n", Time1, Pontos1, Pontos2, Time2);
}

void VerificaMudanca(int Pontos1, int Pontos2, int mud)
{
if ( (Pontos1+Pontos2)%mud == 0)
  {
  puts("Atencao! mudanca de quadra! Digite uma tecla para continuar" );
  getch();
  }
}

void FimdeSet(char *Time1, char*Time2, int Pontos1, int Pontos2)
{
puts("FIM DE SET!");
if(Pontos1>Pontos2)
  printf("%s",Time1);
else
  printf("%s" ,Time2);
puts(" ganhou o set!");
puts("Placar final: ");
MostraPlacar(Time1,Time2,Pontos1,Pontos2);
}

void main()
{
char *Nome[2];
int Equipe1[200], Equipe2[200];
int Set, Mudanca, Saque, Ponto, Dif;
clrscr();
puts("Digite os nomes dos paises:");
gets(Nome[0]);
flushall();
gets(Nome[1]);
puts("Digite a quantidade de pontos do set (15/21):");
scanf("%d",&Set);
Mudanca = Set/3;
```

```
Equipe1[0] = 0;
Equipe2[0] = 0;
Saque = 0;
clrscr();
do
  {
  /* Exibe o placar atual */
  puts("Placar atual:");
  MostraPlacar(Nome[0], Nome[1], Equipe1[Saque], Equipe2[Saque]);
  if (Saque != 0)
            VerificaMudanca(Equipe1[Saque], Equipe2[Saque], Mudanca);
  Saque++;
  puts("Digite a equipe que marcou ponto (1/2):" );
  scanf("%d",&Ponto);
  if (Ponto==1)
            {
            Equipe1[Saque] = Equipe1[Saque-1]+1;
            Equipe2[Saque] = Equipe2[Saque-1];
            }
  else
            {
            Equipe1[Saque] = Equipe1[Saque-1];
            Equipe2[Saque] = Equipe2[Saque-1]+1;
            }
  Dif = abs(Equipe1[Saque] - Equipe2[Saque]);
  clrscr();
  }
while(((Equipe1[Saque]<Set) && (Equipe2[Saque] < Set)) || (Dif < 2)  );
FimdeSet(Nome[0],Nome[1],Equipe1[Saque],Equipe2[Saque]);
getch();
}
```

6.9 Exercícios propostos

0. Escreva uma função recursiva que retorne o maior elemento de um vetor.

1. Escreva uma função que exiba as componentes de um vetor na ordem inversa daquela em que foram armazenadas.

2. Um vetor é *palíndromo* se ele não se altera quando as posições das componentes são invertidas. Por exemplo, o vetor v = {1, 3, 5, 2, 2, 5, 3, 1} é palíndromo. Escreva uma função que verifique se um vetor é palíndromo.

3. Escreva uma função que receba um vetor e o decomponha em dois outros vetores, um contendo as componentes de ordem ímpar e o outro contendo as componentes de ordem par. Por exemplo, se o vetor dado for v = {3, 5, 6, 8, 1, 4, 2, 3, 7}, o vetor deve gerar os vetores u = {3, 6, 1, 2, 7} e w = {5, 8, 4, 3}.

4. Escreva uma função que decomponha um vetor de inteiros em dois outros vetores, um contendo as componentes de valor ímpar e o outro contendo as componentes de valor par. Por exemplo, se o vetor dado for v = {3, 5, 6, 8, 1, 4, 2, 3, 7} a função deve gerar os vetores u = {3, 5, 1, 3, 7} e w = {6, 8, 4, 2}.

5. Um *vetor* do R^n é uma n-upla de números reais v = {x_1, x_2, ..., x_n}, sendo cada x_i chamado de *componente*. A *norma* de um vetor v = {x_1, x_2, ..., x_n} é definida por $\sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$. Escreva uma função que receba um vetor do R^n, n dado, e forneça sua norma.

6. O *produto escalar* de dois vetores do R^n é a soma dos produtos das componentes correspondentes. Isto e, se u = {x_1, x_2, ..., x_n} e v = {y_1, y_2, ..., y_n}, o *produto escalar* é $x_1.y_1 + x_2.y_2 \ldots + x_n.y_n$. Escreva uma função que receba dois vetores do R^n, n dado, e forneça o produto escalar deles.

7. A *amplitude* de uma relação de números reais é a diferença entre o maior e o menor valores da relação. Por exemplo, a *amplitude* da relação 5, 7, 15, 2, 23 21, 3, 6 é 23 - 2 = 21. Escreva uma função que receba uma relação de números e forneça sua *amplitude*.

8. O *desvio padrão* de uma relação de números reais é a raiz quadrada da média aritmética dos quadrados dos desvios (ver exemplo 2, seção 6.4). Escreva uma função que receba uma relação de números

reais e forneça o seu *desvio padrão*.

9. Escreva uma função que forneça as componentes distintas de um vetor dado. Por exemplo, se o vetor dado for v = {3, 2, 1, 3, 4, 1, 5, 5, 2} a função deve fornecer v = {3, 2, 1, 4, 5}.

10. No capítulo 2 foi pedida uma função para extrair o algarismo da casa das unidades de um inteiro dado. Aparentemente esta questão não tem interesse prático. Vejamos um problema cuja solução depende deste problema. Algumas empresas que realizam sorteios de prêmios entre seus clientes o fazem através dos sorteios da loteria federal, sendo ganhador o número formado pelos algarismos das casas das unidades dos números sorteados no cinco prêmios da referida loteria. Por exemplo, se o sorteio da loteria federal deu como resultado os números 2345**1**, 0023**4**, 1123**6**, 0123**5** e 2345**2**, o prêmio da tal empresa seria dado ao cliente que possuísse o bilhete de número **14652**. Escreva uma função que receba os números sorteados pela loteria federal e forneça o número que ganhará o prêmio de acordo com as regras acima.

11. Escreva uma função que insira um valor dado num vetor numa posição dada. Por exemplo, se o vetor for v = {3, 8, 5, 9, 12, 3}, o valor dado for 10 e a posição dada for 4, a função deve fornecer v = {3, 8, 5, **10**, 9, 12, 3}.

12. Escreva uma função que insira um valor dado num vetor ordenado de modo que o vetor continue ordenado. Por exemplo, se o vetor dado for v = {2, 5, 7, 10, 12, 13} e o valor dado for 6, a função deve fornecer o vetor v = {2, 5, **6**, 7, 10, 12, 13}.

13. Escreva uma função que delete uma componente de ordem dada de um vetor dado. Por exemplo, se o vetor dado for v = {2, 5, 7, 10, 12, 13} e a componente a ser deletada for a de ordem 4, programa deve fornecer o vetor v = {2, 5, 7, 12, 13}.

14. Escreva uma função que, dadas duas relações de números, cada uma delas com números distintos, forneça os números que aparecem nas duas listas. Por exemplo, se as relações forem u = {9, 32, 45, 21, 56, 67, 42, 55} e w = {24, 42, 32, 12, 45, 11, 67, 66, 78}, a função deve fornecer o vetor v = {32, 45, 67, 42}.

15. Escreva uma função que, dado um vetor ordenado, forneça a maior diferença entre duas componentes consecutivas, fornecendo também as ordens das componentes que geraram esta maior diferença. Por exemplo, se o vetor dado for v = {3, 5, 9, 16, 17, 20, 26, 31}, a função deve fornecer como maior diferença o valor 7 (16 - 9) e as ordens 4 e 3.

15'. Imagine que as inflações mensais ocorridas num certo país no período de 01/2000 a 12/2008 estejam armazenadas num vetor. Escreva uma função que determine os meses e os respectivos anos em que ocorreram a maior inflação do período.

16. Uma avaliação escolar consiste de 50 questões objetivas, cada uma delas com 5 opções, v = {1, 2, 3, 4 e 5}, sendo apenas uma delas verdadeira. Escreva uma função que receba a sequência de respostas corretas, o *gabarito*, e corrija um cartão-resposta dado.

17. Escreva uma função que forneça o valor numérico de um polinômio P(x) dado, para um valor de x dado. Por exemplo, se o polinômio dado for $P(x) = x^3 + 2x - 1$ e o valor de x dado for 2, a função deve fornecer $P(2) = 2^3 + 2x2 - 1 = 11$.

18. O(s) valor(es) de maior frequência de uma relação de valores numéricos é(são) chamado(s) *moda* da relação. Escreva uma função que receba uma relação de notas escolares maiores do que zero e menores do que ou iguais a 10, com uma casa decimal, e forneça a(s) moda(s) desta relação. Por exemplo, se a relação de notas for v = {8,0; 3,5, 4,5; 8,0; 6,0; 4,5; 6,0; 3,5; 2,5; 6,0; 9,0} a função deve fornecer o valor 6,0 (frequência 3).

19. Escreva uma função que receba um número inteiro *n* e forneça o número formado pelos algarismos de *n* escritos na ordem inversa. Por exemplo, se o número dado for 3876, a função deve fornecer 6783.

20. A matemática prova que a conversão de um número do sistema decimal para o sistema binário pode ser feita através de divisões sucessivas do número e dos quocientes sucessivamente obtidos por 2, sendo então o número binário dado pela sequência iniciada por 1 e seguida pelos restos obtidos nas divisões sucessivas, na ordem inversa em que são obtidos. Por exemplo, para se converter 22 do sistema decimal para o sistema binário temos: 22 % 2 = 0; 11 % 2 = 1; 5 % 2 = 1; 2 % 2 = 0 e, portanto, $22 = (10110)_2$. Escreva uma função que converta um número positivo dado no sistema decimal de numeração para o sistema binário, usando o algoritmo acima.

21. O exercício 10 da seção 4.5 solicitava uma função que determinasse a *decomposição em fatores primos*, fornecendo os fatores primitivos e suas respectivas *multiplicidades*. Na ocasião os fatores primos e suas multiplicidades eram apenas exibidos não sendo armazenados. Modifique a função referida para que os fatores primos e as suas multiplicidades sejam armazenados, antes de serem exibidos.

22. A Universidade Federal de Alagoas adota o sistema de verificação de aprendizagem listado no exemplo 5 da seção 3.4, com o adendo de que terá direito a uma *reavaliação* um aluno que obtiver uma nota inferior a 7,0 em algum bimestre. Neste caso, a nota obtida na reavaliação substitui a menor das notas bimestrais obtidas. Escreva uma função que, recebendo as notas das avaliações bimestrais e, se for o caso, a nota da reavaliação e, se for o caso, a nota da prova final, forneça a média final de um aluno da UFAL e a

sua condição em relação à aprovação.

23. Escreva uma função que forneça a *transposta* de uma matriz dada.

24. Um dos métodos para a se estudar as soluções de um *sistema linear de n equações a n incógnitas* aplica *operações elementares sobre as linhas da matriz dos coeficientes*, sendo a permuta de duas linhas uma destas operações elementares. Escreva uma função que permute as posições de duas linhas de uma matriz dadas.

25. Uma matriz quadrada é dita *triangular* se os elementos situados acima de sua diagonal principal são todos nulos. Escreva uma função que receba uma matriz quadrada e verifique se ela é *triangular*.

26. O exemplo 4 deste capítulo apresentou uma função para armazenar uma matriz simétrica. Este exercício quer algo contrário: escreva uma função que verifique se uma matriz dada é simétrica.

27. Escreva uma função que determine o produto de duas matrizes.

28. Escreva uma função que determine as médias de cada uma das linhas de uma matriz. Por exemplo,

se a matriz dada for $\begin{pmatrix} 3 & 7 & 4 & 6 \\ 5 & 4 & 5 & 4 \\ 2 & 6 & 5 & 1 \end{pmatrix}$ a função deve fornecer a matriz $\begin{pmatrix} 3 & 7 & 4 & 6 & 5,0 \\ 5 & 4 & 5 & 4 & 4,5 \\ 2 & 6 & 5 & 1 & 3,5 \end{pmatrix}$.

29. Escreva um programa que determine o menor valor de cada uma das linhas de uma matriz dada, fornecendo o índice da coluna que contém este menor valor. Por exemplo, se a matriz dada for

$\begin{pmatrix} 3 & 7 & 4 & 6 \\ 5 & 2 & 5 & 4 \\ 2 & 6 & 5 & 1 \end{pmatrix}$, a função deve fornecer uma tabela do tipo

Linha	Menor valor	Coluna
1	3	1
2	2	2
3	1	4

Uma função como esta poderia receber os preços de diversos produtos praticados por vários supermercados e forneceria, para cada produto, o menor preço e o supermercado que pratica este melhor preço.

30. No exemplo 4 da seção anterior vimos como armazenar uma matriz simétrica. Na prática, uma matriz deste tipo ocorre, por exemplo, numa tabela de distâncias entre cidades, como a seguinte tabela, que dá as distâncias aéreas, em *km*, entre as capitais dos estados nordestinos (Aracaju, Fortaleza, João Pessoa, Maceió, Natal, Recife, Salvador, São Luís, Teresina).

	A	F	JP	M	N	R	S	SL	T
A	0	812	438	210	550	398	267	1218	1272
F	812	0	562	730	444	640	1018	640	432
JP	418	562	0	284	144	110	758	1208	987
M	210	730	294	0	423	191	464	1220	1126
N	550	414	144	423	0	252	852	1064	843
R	398	640	118	191	252	0	654	1197	935
S	267	1018	758	464	852	654	0	1319	1000
SL	1218	640	1208	1220	1064	1197	1319	0	320
T	1272	432	987	1126	843	935	1000	320	0

Imagine que uma companhia de transporte aéreo estabeleça que uma viagem entre duas cidades que distem mais de 400 Km deve ter uma escala. Escreva um programa que armazene uma tabela das distâncias aéreas entre *n* cidades e, dadas duas cidades, determine, se for o caso, a cidade em que deve se realizar uma escala para que o percurso seja o menor possível. Por exemplo, nas condições estabelecidas, a viagem entre Maceió e São Luís deve ter uma escala em Fortaleza (o percurso Maceió/Fortaleza/São Luís é de 1370 Km; o percurso, por exemplo, Maceió/Recife/São Luís é de 1388 Km).

31. (Problema não trivial) Utilizando uma função recursiva, escreva um programa que gere as combinações dos números 1, 2, ..., *n* com taxa *k*, *n* e *k* dados. Por exemplo, se n = 5 e k = 3, o programa deve gerar as combinações

1, 2, 3

1, 2, 4
1, 2, 5
1, 3, 4
1, 3, 5
1, 4, 5
2, 3, 4
2, 3, 5
3, 4, 5

Observação

Propostas de soluções dos exercícios propostos podem ser solicitadas através de mensagem eletrônica para jaime@ccen.ufal.br com assunto RESPOSTAS LIVRO C, anexando o formulário abaixo devidamente preenchido.

Nome	**Categoria[1]**	**Instituição[2]**	**Curso[2]**	**Cidade/Estado**

[1]Categoria: docente, estudante, autodidata
[2]Se docente ou estudante

7 Pesquisa e ordenação

7.1 Introdução

Neste capítulo, discutiremos dois problemas clássicos de computação. O primeiro deles, *pesquisa*, *busca* ou *consulta*, consiste em se verificar se um dado valor está armazenado num vetor (ou num *campo* de um *registro* de um *arquivo*, como veremos no capítulo 9).

São vários os exemplos de pesquisas em computação. Uma busca por páginas da *internet* que contenham um determinado assunto; uma busca no Registro Nacional de Veículos Automotores (RENAVAM) na tentativa de se encontrar o nome do proprietário do veículo de uma placa dada; uma busca nos registros da Receita Federal a respeito de um CPF dado.

O segundo problema é conhecido como *ordenação* ou *classificação* (introduzido superficialmente no capítulo 3) consiste em se colocar numa ordem preestabelecida uma relação de valores. No capítulo referido, mostramos como ordenar uma relação contendo três valores. Neste capítulo, apresentaremos algoritmos para ordenar uma lista com qualquer número de valores. A ordenação de uma relação é realizada para que a leitura dos resultados seja facilitada ou para que, como veremos abaixo, pesquisas sejam realizadas com mais eficiência. Um exemplo prático da necessidade da ordenação ocorre na confecção da lista dos aprovados num concurso vestibular. Algumas universidades divulgam esta lista com os nomes dos aprovados em ordem alfabética e outras em ordem de classificação. Tanto num caso como no outro há necessidade de ordenação.

7.2 Pesquisa sequencial

O método de busca de mais fácil compreensão é o que temos utilizado até agora e é chamado *pesquisa sequencial*. Este método consiste em se percorrer, a partir da componente zero, todo o vetor comparando-se o valor de cada componente com o valor pesquisado. Naturalmente, a pesquisa se encerra quando o valor pesquisado é encontrado ou quando se atinge o final do vetor, significando, neste caso, que a pesquisa não foi bem sucedida.

A função abaixo pesquisa, numa relação de inteiros armazenada em *v*, um inteiro passado para o parâmetro *x*. Observe que o parâmetro *t* receberá a quantidade de elementos da relação e que a função retornará a posição do valor procurado na relação, se a pesquisa for bem sucedida, e -1 se o valor procurado não for encontrado.

```
int PesqSeq(int *v, int t, int x)
{
int i;
i = 0;
while ((v[i] != x) && (i < t))
   i = i + 1;
if (i == t)
  return -1;
else
  return i + 1;
}
```

7.3 Pesquisa binária

É muito fácil perceber que o método da pesquisa binária é bastante ineficiente: imagine que este método fosse utilizado para se pesquisar a palavra *zumbaia* num dicionário da língua portuguesa (a propósito, *zumbaia* significa cortesia exagerada; cumprimento ruidoso e servil).

Quando a relação está ordenada, existe um método de busca, chamado *pesquisa binária*, bem mais eficiente do que a pesquisa sequencial: compara-se o elemento pesquisado com a componente "central" da relação; se forem iguais, a pesquisa é encerrada com sucesso; se o elemento pesquisado for menor que a

componente central repete-se a pesquisa em relação à "primeira metade" da relação; se o elemento pesquisado for maior repete-se a pesquisa em relação à "segunda metade" da relação. Por exemplo, uma pesquisa do número 7 na relação {1, 3, 4, 5, 6, 8, 10, 11, 12, 15, 18, 19, 20, 21, 22, 25, 26} começaria comparando-se 7 com 12; como 7 < 12, pesquisa-se 7 na relação {1, 3, 4, 5, 6, 8, 10, 11}; para isto compara-se 7 com 5 e, como 7 > 5, pesquisa-se este valor na relação {6, 8, 10, 11}; pesquisa-se na relação {6, 8}; pesquisa-se em {6} e conclui-se que 7 não está relação.

```
int PesqBinaria(int *v, int t, int x)
{
int i, Central;
i = 0;
Central = t/2;
while ((x != v[Central]) && (i <= t))
  {
  if (x < v[Central])
        t = Central - 1;
  else
        i = Central + 1;
  Central = (i + t)/2;
  }
if (i > t)
  return (-1);
else
  return(Central);
}
```

A pesquisa binária também é importante no desenvolvimento da lógica de programação pelo fato de que é uma função que pode ser implementada recursivamente, sem que a implementação recursiva seja menos eficiente do que a não recursiva. Para perceber a recursividade basta ver que a mesma pesquisa se repete, sendo que, em cada repetição, o vetor pesquisado tem alterado a posição da sua última componente ou da sua primeira componente.

```
int PesqBinRec(int *v, int i, int t, int x)
{
int Central;
Central = (i + t)/2;
if (v[Central] == x)
  return (Central + 1);
else
  if (t < i)
        return (-1);
  else
        if (x < v[Central])
              PesqBinRec(v, i, Central - 1, x);
        else
              PesqBinRec(v, Central + 1, t, x);
}
```

7.4 Ordenação

O *SelecSort*

O algoritmo *SelectSort* consiste em se selecionar, sucessivamente, o maior elemento, o segundo maior elemento, o terceiro maior elemento, etc., e, após cada seleção, armazenar o valor selecionado num vetor auxiliar na posição que mantém o tal vetor auxiliar ordenado. Por exemplo, se se pretende a ordenação em

ordem crescente, o "primeiro maior valor" é armazenado na última posição do vetor auxiliar; o "segundo maior valor" é armazenado na penúltima posição do vetor auxiliar e assim sucessivamente. Para que se obtenha o "segundo maior valor" do vetor, excluímos o "primeiro maior valor" atribuindo a esta componente um valor sabidamente menor do que todos os valores armazenados no vetor. Por exemplo, se os valores do vetor são positivos pode-se atribuir -1 a cada componente já selecionada e já armazenada no vetor auxiliar.

Para exemplificar o método, vamos ordenar o vetor v = {5, 2, 7, 1, 8}. Basta percorrer o vetor 5 vezes selecionando sucessivamente 8, 7, 5, 2 e 1 e realizando as seguintes atribuições:

1. Aux = { , , , , 8}
   v = {5, 2, 7, 1, -1}
2. Aux = { , , , 7, 8}
   v = {5, 2, -1, 1, -1}
3. Aux = { , , 5, 7, 8}
   v = {-1, 2, -1, 1, -1}
4. Aux = { , 2, 5, 7, 8}
   v = {-1, -1, -1, 1, -1}
5. Aux = {1, 2, 5, 7, 8}
   v = {-1, -1, -1, -1, -1},

Para finalizar, basta armazenar nas componentes de *v* as componentes de *Aux*.

```
void MaiorElemento(int *v, int t, int &m, int &p)
{
int i, Pos;
m = v[0];
Pos = 0;
for (i = 1; i < t; i = i + 1)
  if (v[i] > m)
          {
          m= v[i];
          Pos = i;
          }
p = Pos;
}

void SelectSort(int *v, int t)
{
int i, Pos, Aux[500];
for(i = 0; i < t; i = i + 1)
  {
  MaiorElemento(v, t, Aux[t - 1 - i], Pos);
  v[Pos] = -1;
  }
for (i = 0; i < t; i = i + 1)
  v[i] = Aux[i];
}
```

Observe que, como o parâmetro *m* é passado por referência, a função *MaiorElemento()* já armazena no vetor *Aux* os maiores elementos de *v*, nas suas posições definitivas.

Há uma outra versão do *SelectSort* que prescinde de um vetor auxiliar. Se o vetor contém *k* componentes, esta versão consiste em se comparar a maior dentre as k - 1 primeiras componentes com a componente de ordem *k*, permutando-se suas posições se aquela maior componente for menor do que esta última. Esta operação coloca o maior elemento na última posição do vetor, como desejado. Este raciocínio é repetido no vetor das k - 1 primeiras componentes e assim sucessivamente.

```
void SelectSortVersao2(int *v, int t)
{
int Pos, k, m;
k = t - 1;
while (k > 0)
```

```
    {
    MaiorElemento(v, k, m, Pos);
    if (v[k] < v[Pos])
        {
        v[Pos] = v[k];
        v[k] = m;
        }
    k--;
    }
}
```

O *BubbleSort*

O algoritmo *BubbleSort* consiste em se percorrer o vetor a ser ordenado várias vezes, comparando-se cada elemento com o seguinte, permutando suas posições se eles não estiverem na ordem pretendida. Assim, cada vez que o vetor é percorrido o maior (ou o menor) elemento ainda não ordenado é colocado na sua posição de ordenação definitiva. Naturalmente, o vetor será percorrido até que não haja mais trocas a se fazer, quando então ele estará ordenado. Por exemplo, se o vetor a ser ordenado em ordem crescente for v = {5, 1, 9, 3, 7, 2}, teríamos as seguintes configurações para *v*, de acordo com a ordem de percurso:

Percurso	v
0	{5, 1, 9, 3, 7, 2}
1	{1, 5, 9, 3, 7, 2}
	{1, 5, 3, 9, 7, 2}
	{1, 5, 3, 7, 9, 2}
	{1, 5, 3, 7, 2, 9}
2	{1, 3, 5, 7, 2, 9}
	{1, 3, 5, 2, 7, 9}
3	{1, 3, 2, 5, 7, 9}
4	{1, 2, 3, 5, 7, 9}

A seguinte função implementa o algoritmo descrito acima.

```
void BubbleSort(int *v, int t)
{
int j, s, Aux;
do
    {
    s = 1;
    t = t - 1;
    for (j = 0; j < t; j = j + 1)
    if (v[j] > v[j + 1])
        {
        Aux = v[j];
        v[j] = v[j + 1];
        v[j + 1] = Aux;
        s = 0;
        }
    }
while (s == 0);
}
```

Observe que a variável *s* verifica se houve alguma troca para que outro percurso seja realizado. Observe também que o comando t = t – 1 se justifica pelo fato de que no percurso de ordem *i*, i – 1 elementos já estão em suas posições definitivas.

7.5 Exercícios propostos

1. Algumas pessoas acham que são azaradas quando procuram uma ficha numa pilha, sempre tendo receio que a ficha procurada seja uma das últimas da pilha. Uma pessoa que acredite ser assim azarada pode pesquisar a tal ficha pesquisando, sucessivamente, a parte superior e a parte inferior da pilha. Assim, verifica a primeira ficha, em seguida, a última, em seguida, a segunda ficha, em seguida, a penúltima e assim sucessivamente. Escreva uma função que implemente este método de pesquisa.

2. A algoritmo *InsertSort* para ordenação de um vetor *Vet* consiste em se tomar um vetor auxiliar *Aux*, contendo uma única componente Vet[0]. Em seguida, inserem-se as demais componentes de Vet, uma a uma, em *Aux* de modo que *Aux* se mantenha ordenado. Escreva uma função que implemente o *InsertSort*.

3. Escreva uma versão recursiva do *SelectSort*.

Observação

Propostas de soluções dos exercícios propostos podem ser solicitadas através de mensagem eletrônica para jaime@ccen.ufal.br com assunto RESPOSTAS LIVRO C, anexando o formulário abaixo devidamente preenchido.

Nome	Categoria[1]	Instituição[2]	Curso[2]	Cidade/Estado

[1]Categoria: docente, estudante, autodidata

[2]Se docente ou estudante

8. Cadeias de caracteres (*strings*)

8.1 Introdução

Como já sabemos uma declaração do tipo

```
char Cad[10];
```

define um conjunto de dez posições de memória, cada uma delas de um byte, capazes de armazenar variáveis do tipo *char*. Como dissemos de passagem no capítulo anterior, um vetor cujas componentes são do tipo *char* constitui uma *cadeia de caracteres* ou, emprestado do inglês, uma *string*.

Além da quantidade de bytes, o que diferencia a declaração acima da declaração

```
int v[10];
```

é que, enquanto o vetor *v* não pode ser referenciado globalmente com um comando do tipo *scanf("%v", v)* ou *printf("%v", v)* os compiladores C contêm recursos para referência a uma *string* como se ela fosse uma variável simples.

Desta forma, com a declaração acima, podemos ter um programa como o seguinte:

```
#include <stdio.h>
main()
{
char Cad[10];
printf("Digite uma palavra");
scanf("%s", Cad);
printf("A palavra digitada foi: %s ", Cad);
}
```

Vale lembrar que, de acordo com o capítulo 2, "%s" tanto é o *código de conversão* da função *scanf()* para armazenamento de uma cadeia de caracteres como é o *código de formatação* da função *printf()* para exibição de uma *string*. Vale também observar a não colocação do operador de endereço & em *scanf("%s", Cad)*. Para compreender este fato basta lembrar que um vetor é um ponteiro. Ainda vale observar que a tentativa de se armazenar em *Cad* uma cadeia com mais de 10 caracteres não é recusada pelo sistema, mas podem ocorrer armazenamentos indesejados. Para que o tamanho da cadeia não fique limitado, deve-se declarar a cadeia como um ponteiro: char *Cad;.

Também é possível se fazer atribuições explícitas a uma *string*. Por exemplo, poderíamos "sofisticar" o programa acima, escrevendo-o:

```
#include <stdio.h>
main()
{
char Cad[40], Str[40] = "Digite uma palavra";
printf("%s", Str);
scanf("%s", Cad);
printf("A palavra digitada foi: %s\n", Cad);
}
```

A referência explícita a uma *string* é possível pelo fato de que o sistema coloca no final da cadeia o caractere *nulo*, indicado por '\0', tanto numa atribuição (como Str = "Digite uma palavra";) como numa entrada através da função *scanf()*. É este caractere nulo que indica ao sistema o final da *string* para que ele possa processá-la de acordo com o pretendido, como exibi-la através da função *printf()*, por exemplo.

É necessário lembrar que a aposição automática do caractere nulo no final da *string* força se definir *n* numa declaração do tipo

```
char Cad[n];
```

uma unidade maior do que o número de caracteres da cadeia que se pretende armazenar em *Cad*.

Neste formato de declaração, pode-se inicializar uma *string* quando da sua declaração, colocando o valor inicial pretendido entre chaves, que não são necessárias no Turbo C++ 3.0.

```
#include <stdio.h>
main()
{
char Cad[100], Str[20] = {"Digite uma palavra"};
printf("%s \n", Str);
scanf("%s", Cad);
printf("A palavra digitada foi: %s \n", Cad);
}
```

Agora o sistema verifica o número de caracteres da cadeia que se pretende armazenar, recusando, em nível de compilação, se este número for maior do que *n*.

O armazenamento de uma cadeia de caracteres através da função *scanf()* tem uma limitação em função de que esta função considera também (além da digitação de <enter>) o espaço em branco como finalizador da *string*. Isto implica que uma cadeia de caracteres que possua um espaço em branco (o nome de uma pessoa, por exemplo) não possa ser completamente armazenada.

Para estes casos, deve-se utilizar a função *gets()* cujo protótipo está no arquivo *stdio.h* e armazena no argumento com o qual foi ativado uma cadeia de caracteres digitada no teclado. Assim para se armazenar uma frase ou o nome completo de uma pessoa, o programa acima deveria ser modificado para o seguinte programa:

```
#include <stdio.h>
main()
{
char Cad[40];
printf("Digite a frase");
gets(Cad);
printf("A frase digitada foi: %s \n", Cad);
}
```

Repetindo o que já foi dito, é necessário um certo cuidado com execuções sucessivas das funções *scanf()* (para *strings*) e *gets()*. O que acontece é o seguinte: quando os dados estão sendo digitados, eles são armazenados numa área chamada *buffer* e as execuções das funções aqui referidas fazem o sistema armazenar os dados do *buffer* na variável pretendida. Assim, se não estiver vazio, o conteúdo do *buffer* será armazenado por uma próxima ativação de uma destas funções. É prudente, então, "esvaziar" o *buffer* antes de uma chamada de uma segunda chamada de *scanf()* ou de *gets()*, isto podendo ser feitos através da instrução *fflhush(stdin)* (no próximo capítulo, faremos comentários sobre *stdin*).

8.2 Funções de biblioteca para manipulação de cadeias de caracteres

Ao contrário de outras linguagens, C não possui operador que atue com operandos do tipo cadeia de caracteres. Qualquer manipulação de *strings* é feita através de funções da biblioteca padrão de C. Nesta seção, apresentaremos algumas destas funções, cujos protótipos estão no arquivo *string.h*.

a) Determinando o comprimento de uma *string*.

A função de protótipo

```
int  strlen(char *s)
```

retorna o número de caracteres da *string* armazenada em s, sem considerar o caractere nulo.

Por exemplo, a sequência de instruções

```
char *Cad;
int Comp;
Cad = "Universidade Federal de Alagoas";
Comp = strlen(Cad);
```

armazenará na variável *Comp* o valor 31.

Vale observar que o parâmetro de *strlen()* pode ser uma constante: podemos ter um comando

printf("%d", strlen("Brasil"));

b) Comparando duas *strings*.

A comparação entre duas *strings* em relação à ordem alfabética é feita através da função de protótipo

int strcmp(char *s_1, char *s_2);

que retorna a diferença entre os códigos ASCII dos primeiros caracteres diferentes do dois parâmetros, que podem ser constantes. Por exemplo, a chamada *str("Casa", "Caso");* retorna 14, que é a diferença entre os códigos ASCII de 'o' (111) e o de 'a' (97). Naturalmente, se as cadeias são iguais, a função retorna 0 (zero).

c) Convertendo maiusculas para minúsculas e vice-versa.

A conversão das letras de uma *string* de minúsculas para maiusculas é feita através da função de protótipo

char *strupr(char *s);

enquanto que a conversão inversa é feita através da função

char *strlwr(char *s);

podendo o parâmetro ser uma constante.

d) Concatenando uma *string* a outra.

A concatenação de uma cadeia de caracteres a uma outra cadeia é feita através da função

char *strcat(char *s_1, char *s_s);

que retorna a cadeia s_1 acrescida dos caracteres de s_2, que pode ser uma constante. Por exemplo, a sequência de instruções

```
char *Str;
Str = "Computa";
strcat(Str, "dor");
```

armazena em *Str* a cadeia "Computador".

e) Fazendo cópia de uma *string*.

Se s_1 e s_2 foram definidas como *strings* não se pode fazer uma atribuição do tipo $s_1 = s_2$. Se pretendemos armazenar o conteúdo de s_2 em s_1 devemos utilizar a função

char *strcpy(char *s1, char *s2);

que faz uma cópia do conteúdo de s_2 em s_1, podendo s_2 ser uma constante.

f) Copiando parte de uma *string*.

Pode-se copiar os *n* primeiros caracteres de uma *string* através da função

char *strncpy(char *s1, char *s2, int n);

que armazena em s_1 os *n* primeiros caracteres de s_2, podendo este segundo parâmetro ser uma constante. É necessário observar que o caractere nulo não é armazenado, devendo isto ser feito pelo programa.

g) Verificando se uma *string* é subcadeia de outra *string*.

Para se verificar se uma dada cadeia de caracteres está contida em outra cadeia, utiliza-se a função

char *strstr(char *s1, char *s2);

que retorna um ponteiro para a primeira posição a partir da qual s_2 ocorre em s_1, retornando *NULL* se s_2 não está contida em s_1. Por exemplo, a execução do programa

```
#include <stdio.h>
#include <string.h>
main(void)
{
```

```
char *Str1 = "Logica de Programacao", *Str2 = "grama", *p;
p = strstr(Str1, Str2);
if (p != NULL)
	printf("A \"ultima\" substring de %s que contem %s e: %s.\n", Str1, Str2, p);
else
	printf("%s nao esta contida em %s", Str2, Str1);
}
```

exibe na tela a afirmação:

A "ultima" substring de Logica de Programacao que contem grama e gramacao.

h) Convertendo uma *string* em números

Como veremos em exemplos a seguir, muitas vezes é preferível que um dado de entrada seja um vetor de caracteres, mesmo que a função necessite realizar operações matemáticas com ele. A conversão de uma subcadeia de dígitos para os tipos *int*, *long* ou *float*, respectivamente, é feita através das funções

```
int atoi(char *s);
long atol(char *s);
double atof(char *s);
```

cujos protótipos estão no arquivo *stdlib.h*. Estas funções retornam o número (no formato respectivo) correspondente à primeira (da esquerda para direita) subcadeia de *s* que pode ser convertida , retornando 0 (zero) se o primeiro caractere de *s* não for um dígito ou um dos caracteres + e – (se o primeiro caractere for + ou -, para que haja alguma conversão o segundo deve ser um dígito).

8.3 Exemplos Parte VI

1. A função a seguir exclui uma dada quantidade *n* de caracteres a partir de uma posição dada *p*. Para conseguir excluir os *n* caracteres, "trazemos" a *substring* formada pelos últimos caracteres que não serão excluídos mais o caractere nulo para a posição *p*. Naturalmente, se o número de caracteres a serem excluídos for maior do que o disponível, todos os caracteres a partir de *p* serão excluídos. Isto é obtido através da função *strncpy()*.

```
#include <stdio.h>
#include <string.h>
void DeletaCaracteres(char *s, int n, int p)
{
int i,  Comp;
char *Aux;
Comp = strlen(s);
if (p + n <= Comp)
	{
	i = p;
	while (i <= Comp - n)
	     {
	     s[i] = s[i + n];
	     i = i + 1;
	     }
	}
else
	s[p] = '\0';
}
```

2. A próxima função insere uma cadeia s_1 numa cadeia s_2 a partir de uma posição p dada.

```
void Insere(char *s1, char *s2, int p)
{
int i, c1, c2;
```

```
char a[20];
c1 = strlen(s1);
c2 = strlen(s2);
for (i = 0; i < c1 + c2 ; i++)
  if (i < p)
          a[i] = s1[i];
  else
          if (i < p + c2)
                  a[i] = s2[i - p];
          else
                  a[i] = s1[i - c2];
a[i] = '\0';
strcpy(s1, a);
}
```

3. A função *strstr()* comentada acima verifica se uma *string* s_1 está contida numa *string* s_2 mas não retorna a posição a partir da qual s_2 ocorre em s_1. A função abaixo resolve esta questão através da diferença dos comprimentos de s_1 e da cadeia que a chamada *strstr(s1, s2)* retorna.

```
int Pos(char *s1, char *s2)
{
char *Aux;
Aux = strstr(s1, s2);
if (Aux != NULL)
  return(strlen(s1) - strlen(Aux));
else
  return (-1);
}
```

4. Com a função *strncpy()* pode-se copiar os *n* primeiros caracteres de uma *string*. Considerando que um ponteiro de caracteres armazena o endereço da posição de memória do primeiro caractere da *string,* que as posições de memória que armazenam as componentes de um vetor são contíguas e que cada variável do tipo *char* ocupa um *byte*, se s é um ponteiro do tipo *char* e p é um inteiro, s + p será um ponteiro que apontará para o caractere de ordem p. Assim, *strncpy(s1, s2 + p, n)* armazenará em s1 os n caracteres de s2 a partir da posição p.

5. A próxima função converte uma data dada (como uma cadeia de caracteres) no formato americano *mm/dd/aaaa* para o formato brasileiro *dd/mm/aaaa*. O algoritmo usa a função *strncpy()* (com a observação do exemplo anterior) para extrair o dia, o mês e o ano e a função *strcat()* para concatenar na ordem pretendida.

```
void ConverteData(char *s)
{
char *Dia, *Mes, *Ano;
strncpy(Mes, s, 3);
Mes[3] = '\0';
strncpy(Dia, s + 3, 3);
Dia[3] = '\0';
strncpy(Ano, s + 6, 4);
Ano[4] = '\0';
strcat(Dia, Mes);
strcat(Dia, Ano);
strcpy(s, Dia);
}
```

6. Um programa que manipule datas deve possuir uma função que verifique se a data dada era válida. Isto não ocorreria se o valor do mês fosse maior que 12 ou que, por exemplo, se o mês fosse junho e o dia fosse 31.

```
int VerificaData(char s[11])
{
```

```
int i, d, m, a, Verifica;
char Dia[3], Mes[3], Ano[5];
strncpy(Dia, s, 2);
Dia[3] = '\0';
strncpy(Mes, s + 3, 2);
Mes[3] = '\0';
strcnpy(Ano, s + 6, 4);
Ano[4] = '\0';
d = atoi(Dia);
m = atoi(Mes);
a = atoi(Ano);
Verifica = 1;
if ((m <= 12) && (m >= 1) && (d >= 1)  && (d <= 31))
  switch(m)
        {
     case 2:
          if (((a % 4 == 0) && (a % 100 != 0)) || (a % 400 == 0))
               if (d > 29)
                    Verifica = 0;
               else;
          else
               if (d > 28)
                    Verifica = 0;
               break;
     case 4: case 6: case 9: case 11:
          if (d > 30)
               Verifica = 0;
          break;
     }
else
  Verifica = 0;
return(Verifica);
}
```

Vale observar que os comandos *Verifica = 0* poderiam ser substituídos por *return(0)*. Neste caso, o último comando seria *return(1)*.

7. Os compiladores C ignoram espaços em branco digitados num programa. Uma maneira de se tornar isto possível é, antes da compilação, eliminar todos os espaços em branco "supérfluos", ou seja, deixar duas palavras sempre separadas por um único espaço em branco. A função abaixo, utilizando a função *DeletaCaracteres()*, realiza tal ação.

```
void ExcluiBrancosSuperfluos(char *s)
{
int i, NumBrancos;
i = 0;
while (s[i] != '\0')
  {
  NumBrancos = 0;
  while (s[i] == ' ')
     {
     NumBrancos = NumBrancos + 1;
     i = i + 1;
     }
  if (NumBrancos > 1)
     {
     i = i - NumBrancos;
     DeletaCaracteres(s, NumBrancos - 1, i);
     }
```

```
  i = i + 1;
  }
}
```

8. A questão a seguir é bem interessante. Trata-se de um programa que determine o *dígito verificador* de um número de uma conta corrente, de um número de matrícula de um estudante de uma escola, etc. O dígito verificador serve para a prevenção de possíveis erros de digitação. Por exemplo, se a matrícula 30245-7 fosse digitada erroneamente como 39245-7, o erro seria detectado, pois o dígito verificador da conta 39245 seria 6 e não 7. Existem vários métodos para a determinação do dígito verificador. Um deles é dado pelo seguinte algoritmo:

1. Multiplica-se os números correspondentes aos dígitos da conta, da direita para esquerda, por 2, por 3, etc..
2. Soma-se os produtos obtidos no item 1.
3. Determina-se o resto da divisão da soma obtida no item 2 por 11.
4. Subtrai-se de 11 o resto obtido no item 3
5. Se o valor obtido no item 4 for 10 ou 11 o dígito verificado é igual a zero; senão, o dígito é o valor obtido no item referido.

Por exemplo, se o número da conta for 30245, temos

1. 5x2 = 10, 4x3 = 12, 2x4 = 8, 0x5 = 0, 3x6 = 18
2. 10 + 12 + 8 + 0 + 18 = 48
3. Resto(48, 11) = 4
4. 11 - 4 = 7
5. Dígito verificador = 7.

A função abaixo implementa este algoritmo. Observe que foi utilizado *molde* para converter caracteres em inteiros e vice-versa.

```
  int ArmazenaDigitos(char *s, int *v)
{
int i,  Comp;
Comp = strlen(s);
for (i = 0; i < Comp; i = i + 1)
  v[i] = (int) (s[i] - '0');
return(i);
}

char CalculaDigito(char *s)
{
char c;
int t, i, j, Digito, *v;
t = ArmazenaDigitos(s, v);
Digito = 0;
j = 2;
for (i = t - 1; i >= 0; i = i - 1, j = j + 1)
  Digito = Digito + v[i]*j;
Digito = Digito % 11;
Digito = 11 - Digito;
if ((Digito == 10) || (Digito == 11))
  Digito = 0;
c = (char) Digito + '0';
return (c);
}
```

8.4 Exercícios propostos

1. Uma palavra é *palíndroma* se ela não se altera quando lida da direita para esquerda. Por exemplo,

raiar é *palíndroma*. Escreva um programa que verifique se uma palavra dada é palíndroma.

2. Um dos recursos disponibilizados pelos editores de texto mais modernos é a determinação do número de palavras de um texto. Escreva um programa que determine o número de palavras de um texto dado.

3. O exercício 21 do capítulo 6 solicitava um programa que convertesse um número dado no sistema decimal para o sistema binário. Pela limitação do sistema em tratar números inteiros, uma solução que tratasse a conversão como sendo do tipo *long* seria limitada. Escreva um programa para a conversão citada, tratando o valor em binário como uma cadeia de caracteres.

4. Escreva um programa que converta um número do sistema binário, dado como uma cadeia de zeros e uns, para o sistema decimal de numeração.

5. Reescreva a função apresentada no exemplo 8 deste capítulo de tal modo que ele possa, além de gerar dígitos verificadores, verificar se uma conta dada (incluindo o dígito verificador) foi digitada incorretamente, incorreção esta detectada pelo dígito verificador.

6. Os editores de texto possuem um recurso que permite o usuário substituir uma sub-cadeia de um texto por outra cadeia de caracteres. Escreva um programa que realize esta ação numa frase dada.

7. As companhias de transportes aéreos costumam representar os nomes dos passageiros no formato *último sobrenome/nome*. Por exemplo, o passageiro Carlos Drumond de Andrade seria indicado por Andrade/Carlos. Escreva um programa que receba um nome e o escreva no formato acima.

8. As normas para a exibição da bibliografia de um artigo científico, de uma monografia, de um livro texto, etc., exigem que o nome do autor seja escrito no formato *último sobrenome, sequência das primeiras letras do nome e dos demais sobrenomes, seguidas de ponto final*. Por exemplo, Antônio Carlos Jobim seria referido por Jobim, A. C.. Escreva um programa que receba um nome e o escreva no formato de bibliografia.

9. É muito comum que os títulos de documentos como avisos, declarações, atestados, etc., apareçam em letras maiusculas separadas por um espaço em branco. Escreva uma função que receba uma palavra e a retorne no formato acima.

10. Escreva uma função que gere logins para usuários de um sistema de computação baseado na seguinte regra: o login é composto pelas letras iniciais do nome do usuário.

Observação

Propostas de soluções dos exercícios propostos podem ser solicitadas através de mensagem eletrônica para jaime@ccen.ufal.br com assunto RESPOSTAS LIVRO C, anexando o formulário abaixo devidamente preenchido.

Nome	Categoria[1]	Instituição[2]	Curso[2]	Cidade/Estado

[1]Categoria: docente, estudante, autodidata

[2]Se docente ou estudante

9 Estruturas e Arquivos

9.1 O que são estruturas

Um vetor é capaz armazenar diversos valores, com a ressalva de que todos sejam de um mesmo tipo de dado. Um programa que gerencie os recursos humanos de uma empresa manipula dados de tipos diferentes relativos a cada um dos funcionários. Por exemplo, para cada funcionário deve-se ter sua matrícula, seu nome, seu endereço, o cargo que ele ocupa, o número de seus dependentes, o seu salário, a data de admissão, etc. Observe que, nome, matrícula, endereço, data de admissão e cargo que ele ocupa podem ser tratados com variáveis do tipo *string*, porém, como eventualmente haverá necessidade de se efetuar operações aritméticas com eles, o número de dependentes deve ser tratado como do tipo *int* e valor do salário, do tipo *float*.

A utilização de uma variável simples para cada um destes elementos, implicaria, como são vários funcionários, a necessidade de vários vetores, o que poderia atrapalhar a legibilidade e a manutenção do programa, alem de dificultar a possibilidade de armazenamento dos dados em disco, conforme veremos numa seção posterior.

Uma *estrutura* é um conjunto de variáveis, denominadas *campos* ou *membros*, que podem ser de tipos diferentes. É comum se associar um identificador a uma estrutura, chamado *etiqueta* da estrutura, para que se possa definir uma variável deste tipo.

A definição de uma estrutura é feita com a seguinte sintaxe:

```
struct TEstrutura
    {
    Tipo de dado Identificador do campo 1;
    Tipo de dado Identificador do campo 2;
    . . .
    Tipo de dado Identificador do campo n;
  };
```

onde, para cada campo, *Tipo de dado* pode ser qualquer tipo, até mesmo uma estrutura. Se esta declaração for feita fora de qualquer função, a estrutura *TEstrutura* será global e qualquer função do programa pode declarar uma variável capaz de armazenar valores de acordo com os tipos de dados dos campos, isto sendo feito a partir de uma declaração do tipo

```
struct TEstrutura Identificador;
```

Por exemplo, para o programa do exemplo acima poderíamos definir as estruturas

```
struct TEndereco
    {
    char Rua[40];
    char Numero[5];
    char Bairro[10];
    char Cep[9];
    };
struct TEstrutura
    {
    char Matricula[11];
    char Nome[40];
    struct TEndereco Endereco;
    int NumDependentes;
    float Salario;
    char Cargo[8];
    };
```

e então uma função poderia ter uma variável declarada da seguinte forma:

```
struct TEstrutura  Registro;
```

Na verdade, ao se definir *struct TEstrutura* está se definindo um *novo tipo de dado* e isto justifica a definição de uma variável, no caso *Registro*, do tipo de dado *struct TEstrutura*. A linguagem C oferece uma outra forma de se definir um novo tipo de dado. Trata-se da declaração *typedef* que poderia ser utilizado para se definir a estrutura acima da seguinte forma:

```
typedef struct
        {
        char Matricula[11];
        char Nome[40];
        struct TEndereco Endereco;
        int NumDependentes;
        float Salario;
        char Cargo[8];
        }
        TEstrutura;
```

Neste caso, a declaração de uma variável do tipo *TEstrutura* seria feita sem a referência ao tipo *struct*:

```
TEstrutura  Registro;
```

A referência a um campo particular da estrutura se faz com a aposição do identificador da variável estrutura e o identificador do campo separados por um ponto. No exemplo anterior poderíamos ter comandos do tipo

```
Registro.Salario = 4500.00;
scanf("%s", Registro.Matricula);
gets((Registro.Endereco).Rua);
```

sendo os parênteses de (Registro.Endereco).Rua não obrigatórios.

9.2 Exemplos Parte VII

1. Vejamos um programa para controlar as vendas de uma loja, no sentido de, ao final do dia, seja exibida uma lista relacionando todos os produtos vendidos e os vendedores respectivos

```
#include <string.h>
#include <conio.h>
#include <stdio.h>

typedef struct
       {
       char Prod[20];
       char Vend[4];
       float Preco;
       }
       TVenda;

void LeDados(TVenda v[100], int &t)
{
int i = 0;
puts("Digite produto vendedor preço (p/ encerrar digite 0 para o produto");
puts("Produto");  scanf("%s", v[i].Prod);
while (strcmp(v[i].Prod, "0") != 0)
        {
        fflush(stdin);
        puts("Vendedor");
        scanf("%s", v[i].Vend);
        puts("Preço");
        scanf("%f", &v[i].Preco);
        i++;
```

```
        fflush(stdin);
        puts(“Produto”);
        scanf("%s", v[i].Prod);
        }
t = i - 1;
}
void ExibeDados(TVenda v[100], int t)
{
int i;
for (i = 0; i <= t; i++)
        printf("%s        %s %.2f \n", v[i].Prod, v[i].Vend, v[i].Preco);
}
main()
{
TVenda w[100];
int q;
LeDados(w, q);
ExibeDados(w, q);
getch();
}
```

2. Imagine que tenha sido realizada uma pesquisa com 20 pessoas a respeito de salário, idade, número de filhos e sexo. O programa abaixo recebe os dados coletados na pesquisa e fornece a média salarial, a média das idades e o número de mulheres cujo salário é maior R$ 500,00.

```
#include <string.h>
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
typedef struct
        {
        int Salario, Idade, NumFilhos;
        char Sexo;
        }
        TDados;
/*Recebe os dados da coleta e os armazena num vetor de registros&/
void LeDados(TDados v[20], int &i)
{
for (i = 0; i < 20; i++)
        {
        printf("Salario: ");      scanf("%d", &v[i].Salario);
        printf("Idade: ");   scanf("%d", &v[i].Idade);
        printf("Numero de filhos: ");   scanf("%d", &v[i].NumFilhos);
        fflush(stdin);
        printf("Sexo: ");    scanf("%d", &v[i].Sexo);
        }
}
/*Determina os indicadores pretendidos
void Indicadores(TDados v[20], int t, int &MedSal, int &MedIdade, int &NumMulheres, int
&MaiorSal)
{
int i, SomaIdade = 0, SomaSal = 0;
NumMulheres = 0,   MaiorSal = 0;
for (i = 0; i < t; i++)
        {
        SomaSal = SomaSal + v[i].Salario;
```

```
        SomaIdade = SomaIdade + v[i].Idade;
        if (v[i].Sexo == 'F' && v[i].Salario > 500)
                NumMulheres++;
        if (v[i].Salario > MaiorSal)
                MaiorSal =  v[i].Salario;
        }
MedIdade = SomaIdade/t;
MedSal = SomaSal/t;
}
main()
{
TDados w[20];
int q;
int MediaSal, MediaIdade, Mulheres, MaiorSalario;
clrscr();
LeDados(w, q);
clrscr();
ExibeDados(w, q);
Indicadores(w, q, MediaSal, MediaIdade, Mulheres, MaiorSalario);
printf("\nMedia Salarial: %d\nMediaIdade: %d\nNumero de mulheres com salarios superiores a R$
500,00: %d\nMaior Sal rio: %d\n", MediaSal, MediaIdade, Mulheres, MaiorSalario);
getch();
}
```

9.3 O que são arquivos

Até o momento, os dados manipulados pelos nossos programas (dados de entrada, dados gerados pelo programa e resultados do processamento) foram armazenados na memória do computador que, como já foi dito, é uma memória volátil, no sentido de que todas as informações nela armazenadas são perdidas quando a execução do programa é, por qualquer motivo, encerrada.

É evidente que um programa que gerencia os recursos humanos de uma empresa não pode manipular os dados relativos aos funcionários apenas na memória do computador. Isto implicaria, por exemplo, a necessidade de que fossem digitados todos os dados em todas as execuções do programa. É evidente que os dados relativos a cada um dos funcionários da empresa devem estar armazenados, de forma permanente, em um disco, de modo que o programa que gerencia os recursos humanos possa acessá-los em execuções distintas.

Dados e informações reunidos e armazenados num disco constituem um *arquivo* e a linguagem C permite que se manipule arquivos em discos, fornecendo recursos para a realização das operações básicas que podem ser neles executadas: criação de um arquivo, alteração dos dados de um arquivo, exclusão de dados de uma arquivo, inclusão de novos dados no arquivo, exibição (na tela ou em formato impresso) do conteúdo de uma arquivo, etc..

9.4 Arquivos de registros (Arquivos binários)

Os arquivos de uso mais comum na prática de programação em C são os arquivos que armazenam dados oriundos de estruturas. Por exemplo, um sistema que gerencie uma locadora de fitas deve manipular um arquivo que armazene para cada fita, um código, o título do filme, o tema, a data de aquisição, o preço de custo, o valor da locação, etc. No momento da entrada, estes dados podem ser armazenados numa estrutura para serem, em seguida, armazenados num arquivo. Um conjunto de dados relativo a uma fita (neste exemplo) é chamado *registro* e um arquivo em que um conjunto de registros está armazenado é chamado *arquivo de registros*. Nesse caso, os dados são gravados em formato análogo ao formato utilizado para armazenamento em memória. Por esta razão, estes arquivos também são chamados *arquivos binários*.

Uma grande parte das operações que são feitas em arquivos requer a verificação de que o registro que

se pretende manipular está armazenado no arquivo. Isto exige que os registros possuam um campo cujos valores sejam distintos entre si, sendo o valor deste campo um identificador do referido registro. Esta é a função de campos do tipo CPF, matrículas, placas de veículos, etc.. Um campo identificador dos registros de um arquivo é chamado *chave*.

Criando um arquivo de registros

Um arquivo é criado através da função *fopen()*, que possui dois parâmetros do tipo *string* e retorna um ponteiro para uma estrutura pré-definida *FILE* (um ponteiro que aponta para uma estrutura *FILE* é chamado *ponteiro de arquivo*). O primeiro dos parâmetros de *fopen()* fixa o nome com o qual o arquivo será gravado no disco e o segundo parâmetro é a *string* "wb" que indica o formato binário para o arquivo que se está criando. Se, por alguma razão, o arquivo não for criado, a função *fopen()* retorna o ponteiro *NULL*.

Por exemplo, o programa abaixo, cria, no disco da unidade A, um arquivo denominado *Teste.arq*.

```
#include <stdio.h>
FILE *CriaArquivo(char s[12])
{
FILE *p;
p = fopen(s, "wb");
return(p);
}
main()
{
FILE *PontArquivo;
PontArquivo = CriaArquivo("A:\Teste.arq");
if (PontArquivo != NULL)
        printf("Arquivo Teste.arq criado como sucesso");
else
        printf("O arquivo Teste.arq não foi criado");
}
```

A função *fopen()*, além de criar um arquivo gravado no disco associa, através do que ela retorna, um ponteiro para o arquivo referido. Este ponteiro é utilizado para se referenciar o tal arquivo no restante do programa e é chamado *fluxo*. É o caso do ponteiro de arquivo *PontArquivo* do exemplo anterior. Qualquer referência a ele será uma referência ao arquivo *Teste.arq*.

A criação de um arquivo com a ativação de *fopen(NomeArquivo, "wb")* deve ser solicitada com cautela. Se o arquivo de nome *NomeArquivo* existir, a chamada referida "apagará" todo o seu conteúdo. O tipo de cautela necessária veremos a seguir.

Gravando registros em um arquivo

O programa anterior, apenas cria o arquivo no disco não gravando nenhum registro. Se, após a sua execução, procurarmos com o *Windows Explorer* o arquivo *Teste.arq*, encontrá-lo-emos com a indicação de que seu conteúdo tem zero bytes. Ou seja é um arquivo *vazio*. É natural que se crie um arquivo para gravar registros e isto pode ser feito quando da sua criação.

A gravação de registros em um arquivo é feito através da função *fwrite()* que possui quatro parâmetros:

1. Um ponteiro *r* para uma variável do tipo *void* que receberá o endereço da variável do tipo estrutura que contém os dados que se quer armazenar;
2. Um inteiro *n* que receberá o tamanho, em bytes, do registro a ser armazenado;
3. Um inteiro *q* que receberá o número de registros que serão armazenados;
4. Um ponteiro de arquivo *p* que receberá o fluxo associado através da função *fopen()* ao arquivo de disco onde os dados serão armazenados.

Para se criar um arquivo de registros com a estrutura *TRegistro* definida na seção anterior e, logo em seguida, gravar registros no arquivo, teríamos o seguinte programa.

```
#include <stdio.h>
struct TRegistro
  {
  char Mat[4];
  char Nome[40];
  float SalarioBruto;
  };
/*Função que cria um arquivo em disco, deixando-o apto a armazenar dados */
FILE *CriaArquivo(char s[12])
{
FILE *p;
p = fopen(s, "wb");
return(p);
}
/*Função que grava dados armazenados numa estrutura em um arquivo*/
void GravaRegistros(char s[12])
{
FILE *p;
struct TRegistro r;
printf("Matricula (para encerrar, digite matricula 0): ");
fflush(stdin);
gets(r.Mat);
while ((r.Mat)[0] != '0')
        {
        printf("Nome: ");
        fflush(stdin);
        gets(r.Nome);
        printf("Salario bruto: ");
        scanf("%f", &r.SalBruto);
        fwrite(&r, sizeof(r), 1, p);
        printf("Matricula (para encerrar, digite matricula 0): ");
        fflush(stdin);
        gets(r. Mat);
        }
fclose(p);
}
main()
{
FILE *PontArquivo;
char NomeArq[12];
printf("Digite o nome do arquivo");
gets(NomeArq);
PontArquivo = CriaArquivo(NomeArq);
if (PontArquivo != NULL)
        GravaRegistros(NomeArq);
else
        printf("O arquivo %s nao pode ser criado \n", NomeArq);
}
```

A função *fclose()* "fecha" um arquivo para que o sistema operacional possa atualizar a tabela do diretório de arquivos. Observe que esta função tem um parâmetro que receberá o fluxo associado ao arquivo que se pretende fechar. É necessário que todo arquivo "aberto" seja fechado antes do encerramento da execução do programa.

Exibindo o conteúdo de um arquivo

Para se ter acesso ao conteúdo de um arquivo é necessário que este conteúdo seja transferido para memória do computador para, em seguida, ser exibido na tela pela função *printf()* ou impresso por uma impressora através da função *fprintf()* (veremos isto numa seção seguinte). A transferência do conteúdo de um arquivo para memória pode ser feita registro a registro, armazenando cada um deles em uma estrutura, ou através de um conjunto de registros, armazenando-o num vetor de estruturas. Aqui optaremos pela primeira alternativa.

A transferência de registros de um arquivo para a memória é feita através da função *fread()* que, como a função *fwrite()*, possui quatro parâmetros:

1. Um ponteiro para uma variável do tipo *void* que receberá o endereço da variável que armazenará os dados contidos no registro;
2. Um inteiro que receberá o tamanho, em bytes, da estrutura que armazenará o registro na memória;
3. Um inteiro que receberá o número de registros que serão transferidos para a memória (com a opção aqui escolhida este parâmetro sempre será igual a 1);
4. Um ponteiro de arquivo que receberá o fluxo associado, através da função *fopen()*, ao arquivo de disco que contém os registros.

Para que seja possível a aplicação da função *fread()* é necessário que o arquivo esteja "aberto para leitura", o que é feito também através da função *fopen()* agora com segundo parâmetro "rb". Quando é feita uma chamada da função *fopen()* com os argumentos Nome do Arquivo e "rb", o primeiro registro do arquivo fica disponível para leitura (registramos este fato dizendo que o *ponteiro de leitura e gravação* aponta para o primeiro registro).

Considerando-se que após a execução da função *fread()* o ponteiro de leitura e gravação avança automaticamente para o próximo registro, pode-se percorrer todo o arquivo até atingir o seu final, que é fornecido pela função *feof()*. Esta função tem como parâmetro um ponteiro de arquivo e retorna um número diferente de zero quando o ponteiro de leitura e gravação aponta para o final do arquivo.

Por exemplo, pode-se exibir na tela o conteúdo do arquivo gerado acima através da seguinte função.

```
/*Função que exibe na tela o conteúdo de um arquivo */
void ExibeArquivo(char s[12])
{
FILE *p;
struct TRegistro r;
p = fopen(s, "rb");
fread(&r, sizeof(r), 1, p);
while (feof(p) == 0)                    /*Ou, o que é o mais utilizado,  while (!feof(p))*/
    {
    printf("%s \b %s \b %f \n", r.Mat, r.Nome, r.SalarioBruto);
    fread(&r, sizeof(r), 1, p);
    }
fclose(p);
}
```

Verificando a existência de um arquivo

A ativação de *fopen()* no modo "rb" (o segundo parâmetro de *fopen()* é chamado *modo de abertura do arquivo*) permite que se escreva uma função para verificar a existência de um arquivo, que será útil para evitar uma ativação "desastrada" de *fopen()*, já que, como dissemos acima, a ativação desta função no modo "wb" apaga todo o conteúdo do arquivo que possuir o nome passado para o primeiro parâmetro desta função. Isto implica a necessidade de que se tenha cuidado na abertura de um arquivo no modo "wb", pois se for passado um nome de um arquivo que já existe todo o seu conteúdo será perdido. É prudente, portanto, que a abertura de um arquivo no modo aqui discutido seja precedida de uma função que verifique se um arquivo com o nome escolhido já existe. Uma função com este objetivo é bastante simples, pois a função *fopen()* retorna *NULL* se for ativada no modo "rb" com o arquivo que não existe.

```
int ExisteArquivo(char s[12])
```

```
{
FILE *p;
p = fopen(s, "rb");
if (p == NULL)
        return(0);
else
        {
        fclose(p);
        return(1);
        }
}
```

Assim a função *CriaArquivo()* definida anteriormente deveria ser escrita da seguinte forma:

```
FILE *CriaArquivo(char s[12])
{
FILE *p;
p = fopen(s, "rb");
if (p == NULL)
        {
        p = fopen(s, "wb");
        return(p);
        }
else
        printf("\a Arquivo %s já existe!");
}
```

Localizando um registro num arquivo

Uma operação muito comum em arquivos é a verificação de que um determinado registro está nele armazenado. Esta operação é normalmente (como já foi dito no capítulo 7) chamada *consulta*, *pesquisa* ou *busca* e deve ser feita de acordo com o valor da chave do registro ou de um outro campo que, relativamente, identifique o registro. No exemplo que estamos discutindo, a consulta pode ser feita pelo campo *Mat* (de matrícula) ou pelo campo *Nome*. Em geral, a consulta se processa com a localização do registro, a consequente exibição do seu conteúdo e o retorno da posição que ele ocupa no arquivo.

A localização do registro pode ser feita, abrindo-o com *fopen()* e o percorrendo até que o valor da chave seja encontrado; a exibição do seu conteúdo pode ser feita através das funções *fread()* e *fprintf()*, e a posição que ele ocupa no arquivo é fornecida por uma das funções *fgetpos()* e *ftell()* que possuem os protótipos

```
int fgetpos(FILE *p, fpos_t *pos);
long ftell(FILE *p);
```

onde, na primeira, *fpos_t* é um tipo de dado pré-definido.

Nas duas funções, *p* receberá o ponteiro associado ao arquivo onde está se realizando a pesquisa; a posição do registro pesquisado (dada pela ordem do último byte ocupado pelo último campo deste registro) é armazenada na variável cujo endereço for passado para o parâmetro *pos* de *fgetpos()* ou será retornado pela função *ftell()*. Como em vetores, o primeiro byte ocupado pelo primeiro campo do primeiro registro é o de ordem zero.

```
/*Função que verifica se um registro com matricula dada pertence ao arquivo, retornando sua posição no arquivo*/
int ConsultaRegistro(char s[12], char s1[12])
{
FILE *p;
int Achou = 0;
struct TRegistro r;
fpos_t Byte;
```

```
p = fopen(s1, "rb");
fread(&r, sizeof(r), 1, p);
while (!feof(p) && Achou == 0)
        if (strcmp(s, r.Mat) == 0)
                {
                fgetpos(p, &Byte);
                Achou = 1;
                }
        else
          fread(&r, sizeof(r), 1, p);
if (Achou == 0)
        return (-1);
else
        return(Byte);
}
```

Como no nosso exemplo o tamanho da estrutura é de 48 bytes (4 bytes para o campo Mat, 40 para o campo Nome e 4 para o campo SalarioBruto), se o registro pesquisado for o primeiro a função retornará 48, se o registro pesquisado for o segundo, a função retornará retorna 96, se for o terceiro, a função retornará 144 e assim por diante. Se o registro não estiver no arquivo, a função retornará –1.

Quando o registro é encontrado, seu conteúdo está armazenado na estrutura r. Assim, para exibir o conteúdo do registro, basta no comando *if (strcmp(s, r.Mat) == 0)* incluir o comando

```
printf("Matricula: %s \n Nome: %s \n Salario: %f \n", r.Mat, r.Nome, r.SalBruto);
```

Para escrever a função acima com a função *ftell()* bastaria se substituir os comando *fgetpos(p, &Byte)* pelo comando *Byte = ftell(p)*.

Considerando que a instrução *return()* interrompe a execução de uma função, a função acima poderia prescindir da variável *Achou*:

```
int ConsultaRegistro1(char s[12], char s1[12])
{
FILE *p;
struct TRegistro r;
fpos_t Byte;
p = fopen(s1, "rb");
fread(&r, sizeof(r), 1, p);
while (!feof(p))
        if (strcmp(s, r.Mat) == 0)
            {
            fgetpos(p, &Byte);
          return(Byte);
          }
        else
            fread(&r, sizeof(r), 1, p);
return (-1);
}
```

Optamos pela primeira versão pelo fato de que existem linguagens que não possuem instruções do tipo *return()* e, nestas linguagens, teríamos de escrever a função como na versão inicial.

Vale observar que as funções *ConsultaRegistro()* acima utilizam a *pesquisa sequencial*. Se os registros dos arquivos estiverem ordenados pelo campo *Mat* poderíamos ter utilizado a *pesquisa binária*, que, como estudado no capítulo 7, é bem mais eficiente.

Alterando o conteúdo de um registro

Às vezes, há necessidade de que os dados de um registro sejam alterados. No arquivo que estamos

utilizando como exemplo isto poderia ocorrer no caso de uma promoção de um funcionário que implicasse um aumento no seu salário bruto ou no caso de uma funcionária que alterou o seu nome em função de um casamento.

Uma função para alterar os dados de um registro deve, inicialmente, abrir o arquivo para leitura e gravação, o que é feito através da função *fopen()* no modo "rb+". Feito isto, a função deve receber o valor da chave do registro e com este valor chamar a função *ConsultaRegistro()* definida acima para obter a posição do registro pretendido. Tendo esta posição, deve posicionar o ponteiro de leitura e gravação naquele registro e realizar as alterações que são desejadas. Para posicionar o ponteiro de leitura e gravação num determinado registro utiliza-se a função *fsetpos()* cujo protótipo é

```
int fsetpos(FILE *p, fpos_t *pos);.
```

Numa ativação desta função, o parâmetro *p* recebe o ponteiro associado ao arquivo e *pos* recebe a posição do registro, obtido pela função *fgetpos()* ou pela função *ftell()*.

No exemplo que estamos discutindo, podemos alterar o campo *Nome* de um registro de campo *Mat* dado utilizando a seguinte função.

```
/*Função que altera o nome de um registro, dada a matrícula */
void AlteraRegistro(char s[4], char s1[12])
{
char c;
struct TRegistro r;
fpos_t Byte;
int Tam;
FILE *p;
Tam = sizeof(r);
Byte = ConsultaRegistro(s, s1);
if (Byte != -1)
	{
	Byte = Byte – Tam;
	p = fopen(s1, "rb+");
	fsetpos(p, &Byte);
	fread(&r, Tam, 1, p);
	printf("Nome atual: %s \n Altera (S/N)? ", r.Nome);
	fflush(stdin);
	scanf("%c", &c);
	if (toupper(c) == 'S')
			{
		printf("\Digite o novo nome: \n");
		gets(r.Nome);
		fsetpos(p, &Byte);
		fwrite(&r, Tam, 1, p);
		}
	}
else
	printf("\n Registro nao encontrado \n");
fclose(p);
}
```

Observe que, ao contrário das funções anteriores, optamos por armazenar o valor de *sizeof(r)* na variável *Tam*, para evitar várias chamadas dessa função. Observe também que o comando *Byte = Byte – Tam* posiciona o ponteiro no início do registro que se pretende alterar.

Outra função que posiciona o ponteiro de leitura e gravação num registro de posição conhecida é a função *fseek()* que tem o seguinte protótipo:

```
int fseek(FILE *p, long pos, int orig)
```

Aí, *p* receberá o fluxo associado ao arquivo e *pos* indicará a nova posição do ponteiro, a partir da posição dada pelo valor passado para *orig*. O sistema possui três constantes pré-definidas, *SEEK_SET*, *SEEK_CUR* e *SEEK_END* que podem ser passados para o parâmetro *orig*. A primeira toma como origem o

registro zero do arquivo; a segunda, o registro apontado pelo ponteiro de leitura e gravação (*registro corrente)*; a terceira, o final do arquivo. No caso da posição do registro ser obtida por *fgetpos()* ou por *ftell()*, o valor que deve ser passado para *orig* é *SEEK_SET*.

Desta forma, para se escrever a função *AlteraRegistro()* escrita acima utilizando-se a função *fseek()* basta substituir os comandos *fsetpos(p, &Byte)* pelo comando *fseek(p, Byte, SEEK_SET)*.

A constante *SEEK_END*, a função *fseek()* e a função *ftell()* permitem determinar o tamanho, em bytes, de um arquivo. Basta posicionar o ponteiro de leitura e gravação no final do arquivo através de *fseek(p, 0, SEEK_END)* e obter a posição do ponteiro através de *ftell(p)*.

```
int TamanhoArquivo(char *s)
{
FILE *p;
int Tamanho;
p = fopen(s, "rt");
fseek(p, 0, SEEK_END);
Tamanho = ftell(p);
fclose(p);
return(Tamanho);
}
```

Com esta função é possível se determinar o número de registros de um arquivo. Basta dividir o tamanho do arquivo pelo tamanho de cada registro:

```
int NumRegistros(char *s)
{
FILE *p;
struct TRegistro r;
p = fopen(s, "rt");
return(TamanhoArquivo(s)/sizeof(r));
fclose(p);
}
```

Vale observar que, da mesma forma que a chamada de *fseek(p, 0, SEEK_END)* posiciona o ponteiro de leitura e gravação no final do arquivo, *fseek(p, 0, SEEK_SET)* posiciona o tal ponteiro no início do arquivo (existe outra forma de apontar o ponteiro de leitura e gravação para o início do arquivo: *rewind(p)*).

Incluindo novos registros num arquivo

A inclusão de novos registros em um arquivo é feita de forma bastante simples, pois a função *fopen()* ativada no modo "ab+" abre um arquivo e permite que novos registros sejam nele gravados. Naturalmente, a inclusão de um novo registro deve ser precedida da verificação de que o tal registro já está armazenado no arquivo, o que impediria uma nova inclusão. Temos a seguinte sugestão para atingir o objetivo aqui proposto:

```
/* Função que inclui um novo registro num arquivo */
void IncluiRegistro(struct TRegistro r, char s[12])
{
char c;
long Byte;
int Tam;
FILE *p;
Tam = sizeof(r);
Byte = ConsultaRegistro(r.Mat, s);
if (Byte == -1)
	{
	p = fopen(s, "ab+");
	fwrite(&r, Tam, 1, p);
```

```
	}
else
	printf("\n Registro ja cadastrado \n");
fclose(p);
}
```

Excluindo um registro de um arquivo

Outra operação muito utilizada em arquivos é a exclusão de um registro. No nosso exemplo, esta operação seria necessária, por exemplo, na ocasião de um pedido de demissão de um funcionário. Uma possível solução é, após localizar o registro, gravar todos os outros registros num arquivo auxiliar, *Temp*, excluir do disco o arquivo original e renomear o arquivo *Temp* com o nome do arquivo original.

A maioria dos compiladores C excluem um arquivo através da função *remove()* que possui um parâmetro do tipo vetor de caracteres para receber o nome do arquivo a ser removido. Para renomear um arquivo, os compiladores C possuem a função *rename()* que possui dois parâmetros do tipo vetor de caracteres, devendo o primeiro receber o nome atual do arquivo e o segundo receber o novo nome que se pretende.

Dentro do exemplo que estamos estudando, a função abaixo recebendo o valor do campo *r.Mat* e o nome do arquivo, exclui, se a matrícula dada for uma matrícula cadastrada, o registro correspondente.

```
/*Função que exclui um registro de matrícula dada */
void ExcluiRegistro(char s[4], char s1[12])
{
struct TRegistro r;
char c;
long Byte;
int Tam, Reg;
FILE *p, *t;
Tam = sizeof(r);
Byte = ConsultaRegistro(s, s1);
if (Byte != -1)
	{
	p = fopen(s1, "rb");
	Byte = Byte – Tam;
	fsetpos(p, &Byte);
	fread(&r, Tam, 1, p);
	printf("Matricula: %s \b Nome: %s \n", r.Mat, r.Nome);
	printf("Exclui este registro (S/N)? ");
	fflush(stdin);
	scanf("%c", &c);
	if (toupper(c) == 'S')
	    {
	  t = fopen("Temp", "wb");
	  rewind(p);         /*Primeiro registro do arquivo*/
	    Reg = 0;
	    fread(&r, Tam, 1, p);
	        while (!feof(p))
	              {
	              if (Reg != Byte)
	                        fwrite(&r, Tam, 1, t);
	              Reg = Reg + Tam;
	              fread(&r, Tam, 1, p);
	              }
	  fclose(p);
	  fclose(t);
```

```
            remove(s1);
            rename("Temp", s1);
            }
        }
else
        printf("\n Registro nao encontrado \n");
}
```

9.5 Arquivo texto

Outra forma de arquivo que os compiladores C manipulam são os chamados *arquivos textos*. Nestes arquivos, também criados através da função *fopen()*, agora no modo "wt", cadeias de caracteres podem ser armazenadas byte a byte, através do código *ASCII* de cada caractere.

A gravação de texto em um arquivo texto pode ser feita através da função *fprintf()* que, além dos parâmetros da função *printf()*, exige um primeiro parâmetro do tipo fluxo que indicará o arquivo onde o texto será gravado. Por exemplo, o programa abaixo cria um arquivo *Teste.txt* e grava nele a frase *Isto é um teste*.

```
#include <stdio.h>
main()
{
FILE *PontArquivo;
PontArquivo = fopen("Teste.txt", "wt");
fprintf(PontArquivo, "Isto é um teste");
fclose(PontArquivo);
}
```

Após a execução deste programa, qualquer processador de texto que edite textos em *ASCII* (inclusive o *Bloco de Notas* do *Windows*) pode abrir o arquivo *Teste.txt*, sendo o seu conteúdo absolutamente legível.

Na verdade, é possível gravar conteúdos de variáveis e resultados de processamentos em arquivos utilizando-se a função *fprintf()*. Nestes casos, são utilizados os códigos de especificação de formato da função *printf()*. Por exemplo, para se armazenar no arquivo *Teste.txt* uma tabela de raízes quadradas dos cem primeiros inteiros positivos basta se executar o seguinte programa:

```
#include <stdio.h>
#include <math.h>
main()
{
int i;
float r;
FILE *PontArquivo;
PontArquivo = fopen("Teste.txt", "wt");
for (i = 1; i <= 100; i++)
        {
        r = sqrt(i);
        fprintf(PontArquivo, "%d %f \n", i, r);
        }
fclose(PontArquivo);
}
```

A função *printf()* é, de fato, um caso particular da função *fprintf()* cujo primeiro parâmetro é o fluxo pré-definido *stdout*. O fluxo *stdout*, que é omitido em *printf()*, aponta para um arquivo que faz referência ao dispositivo padrão de saída, em geral a tela do monitor. Ou seja, *printf()* "grava" a saída na tela, enquanto *fprintf()* grava a saída no arquivo associado ao fluxo passado para ela. Por exemplo, os comandos *printf("Estou aprendendo a programar em C")* e *fprintf(stdout, "Estou aprendendo a programar em C")* executam ações idênticas: exibem na tela a frase *Estou aprendendo a programar em C*.

Um outro fluxo pré-definido é *stdprn* que aponta para um arquivo que gerencia a relação entre o

sistema e a impressora conectada ao computador. Por exemplo, o programa

```
#include <stdio.h>
main()
{
int i;
for (i = 0; i < 10; i++)
	fprintf(stdprn, "Estou aprendendo a programar em C \n");
}
```

imprime, através da impressora conectada ao computador, dez vezes a frase Estou aprendendo a programar em C, uma vez em cada linha.

Um terceiro fluxo pré-definido é *stdin* que aponta para um arquivo que administra a relação do sistema com o dispositivo de entrada padrão, em geral o teclado. Isto explica a chamada de *fflush(stdin)* comentada no capítulo 5.

Exibindo um arquivo texto

A biblioteca da linguagem C dispõe de uma função capaz de "ler" uma linha de um arquivo texto, armazenando-a num ponteiro de caracteres. Trata-se da função de protótipo

```
char *fgets(char *s, int n, FILE *p);
```

que lê uma quantidade *x* de caracteres do arquivo associado a *p* e os armazena em *s*, como uma *string*. A quantidade de caracteres *x* é inferior ou igual a *n*, sendo inferior quando uma marca de fim de linha é atingida, ou seja, quando o caractere indicado por \n é encontrado. Quando o fim de arquivo é alcançado, a função retorna NULL.

Com *fgets()* é possível exibir o conteúdo de um arquivo texto com a seguinte função:

```
#include <stdio.h>
void ExibeArquivoTexto(char *s)
{
FILE *p;
char *Linha, *Fim;
p = fopen(s, "rt");
Fim = fgets(Linha, 80, p);
while (Fim != NULL)
	{
	printf("%s", Linha);
	Fim = fgets(Linha, 80, p);
	}
}
```

Se este arquivo adicionado da função

```
main()
{
char *NomeArq;
puts("Digite o nome do arquivo");
scanf("%s", NomeArq);
ExibeArquivoTexto(NomeArq);
}
```

for gravado com o nome *ExibText.c*, sua execução para a entrada *exibtext.c* exibe o seu próprio conteúdo.

Utilizando um arquivo texto como entrada de dados

É possível utilizar um arquivo texto para obter a entrada de um programa. Imaginemos que o arquivo texto *card.txt* contivesse o cardápio da lanchonete referida num exemplo do capítulo 4. Ou seja, suponhamos que o conteúdo do arquivo *card.txt* fosse o seguinte:

```
101 Refrigerante 1.20
102 Suco 1.00
103 Sanduíche 2.50
104 Salgado 1.00
105 Torta 2.00
```

O programa abaixo lê este arquivo e armazena num vetor de estruturas permitindo o gerenciamento dos pedidos dos clientes.

```
#include <math.h>
#include <conio.h>
#include <stdio.h>
#include <string.h>

typedef struct
        {
  int Cod;
  char Prod[30];
  float Preco;
        }TItem;

typedef struct
        {
        int Cod;
        int Quant;
        }TPedido;

TItem Cardapio[100];
TPedido Pedido[100];

/*Função para “separar” os componentes do cardápio*/
void ProcessaItem(char *s, int &Codigo, char *Produto, float &Pr)
{
char a[10];
int c = strlen(s);
Codigo = atoi(s);
strncpy(Produto, s + 4, c-9);
strncpy(a, s + c - 5, 4);
Pr = atof(a);
}

/*Funcao para ler o cardápio do arquivo texto*/
int LerCardapio(char *s)
{
FILE *p;
int i = 0;
char *Item, *FimDeLinha;
p = fopen(s, "rt");
FimDeLinha = fgets(Item, 80, p);
while (FimDeLinha != NULL)
  {
  ProcessaItem(Item, Cardapio[i].Cod, Cardapio[i].Prod,  Cardapio[i].Preco);
  i++;
  FimDeLinha = fgets(Item, 80, p);
  }
return i;
}
```

```
/*Função para exibir o cardápio*/
void ExibeCardapio(TItem v[100], int t)
{
int i;
printf("COD                    ESPECIFICACAO                  PRECO\n");
for (i = 0; i < t; i++)
  printf("%d        %-20s     %.2f\n", v[i].Cod, v[i].Prod, v[i].Preco);
}
/* Funcao para exibir pedido*/
void ExibePedidos(int n)
{
int i, Ind;
printf("COD  ESPECIFICACAO                  QUANT   PRECO\n");
for(i=0; i<n; i++)
        {
  Ind = Pedido[i].Cod - 101;
  printf("%3d     %-30s   %5d     R$   %.2f\n",Pedido[i].Cod,   Cardapio[Ind].Prod,   Pedido[i].Quant,
Cardapio[Ind].Preco*Pedido[i].Quant);
  }
}
void main()
{
int i, NumItens;
float Total;
clrscr();
NumItens = LerCardapio("card.txt");
ExibeCardapio(Cardapio, NumItens);
Total = 0;
i = 0;
puts("Digite o codigo do produto desejado (0 para encerrar):");
scanf("%d",&Pedido[i].Cod);
while(Pedido[i].Cod != 0)
  {
  if( (Pedido[i].Cod > 100) &&(Pedido[i].Cod < 106) )
          {
            puts("Digite a quantidade:");
            scanf("%d",&Pedido[i].Quant);
            Total = Total + Pedido[i].Quant * Cardapio[Pedido[i].Cod - 101].Preco;
            i++;
            }
        else
          {
            puts("Erro! O código informado não está cadastrado!");
            puts(“Digite uma tecla para continuar”);
            getch();
            }
  puts("Digite o codigo do produto desejado (0 para encerrar):");
  scanf("%d",&Pedido[i].Cod);
  }
puts("\nTotal dos pedidos:");
ExibePedidos(i);
printf("\nValor total dos pedidos: R$ %.2f", Total);
getch();
}
```

9.6 Exercícios propostos

1. Escreva uma função *main()* que, através de um menu de opções, utilize as funções estudadas neste capítulo e que, portanto, seja capaz de

a) criar e gravar dados num arquivo de registros;
b) exibir o conteúdo de um arquivo;
c) alterar dados de um registro de um arquivo;
d) incluir novos registros em um arquivo;
e) excluir um registro de um arquivo.

2. Escreva uma função que reúna dois arquivos de registros de mesma estrutura em um terceiro arquivo.

3. Escreva um programa que, dado um arquivo cujos registros possuem os campos *Nome* (do tipo vetor de *strings*) e *Salario* (do tipo *float*), gere um arquivo dos registros cujo campo *Salario* é maior que 5.000,00.

4. Escreva uma função que inclua um registro dado num arquivo de registros ordenado por um campo *Mat* de modo que o arquivo se mantenha ordenado, sem utilizar um arquivo auxiliar.

5. Escreva um programa que, dados dois arquivos ordenados por um campo *Mat*, gere um terceiro arquivo também ordenado.

6. Escreva um programa que dados dois arquivos cujos registros têm os campos *char Cpf[12]* e *char Nome[40]* gere um arquivo contendo os registros que pertencem aos dois arquivos.

7. Escreva uma função que troque as posições de dois registros de um arquivo.

8. Escreva uma função que ordene um arquivo, cujos registros têm os campos *char Cpf[12]* e *char Nome[40]*, em relação ao campo *Cpf*.

9. Escreva uma função que exclua os comentários de um programa da linguagem C.

10. Para uma pesquisa relativa aos hábitos de estudo dos alunos do Curso de Ciência da Computação da Universidade Federal de Alagoas, foi idealizado um sistema baseado numa estrutura *TDados,* com os campos *char* Nome[40]; *char* Sexo; *int* NumHoras;, onde *Nome* e *Sexo* têm finalidades óbvias (Sexo recebe F ou M) e *NumHoras* recebe o número de horas diárias de estudo do pesquisado.

x) Escreva uma função que armazene os dados coletados (quantidade de alunos pesquisados não conhecido a priori) num vetor de estruturas.

y) Escreva uma função que retorne os nomes dos/as alunos/as que dedicam mais horas diárias ao estudo.

11. Imagine que o arquivo texto *Notas.txt* contém as notas finais dos alunos da disciplina Programação 1do Curso de Ciência da Computação da UFAL como abaixo

Gisele Bachen 9.9
Juliana Raes 9.8
Ana Paula Ardósia 9.0
Rodrigo Sentouro 5.5
. . .

no qual a coluna dos nomes está alinhada à esquerda, as notas têm sempre uma casa decimal, podendo ter ocorrido nota 10.0, e não há espaços em branco após cada nota. Escreva uma função que receba o nome do arquivo e retorne as médias das notas.

Observação

Propostas de soluções dos exercícios propostos podem ser solicitadas através de mensagem eletrônica para jaime@ccen.ufal.br com assunto RESPOSTAS LIVRO C, anexando o formulário abaixo devidamente preenchido.

Nome	Categoria[1]	Instituição[2]	Curso[2]	Cidade/Estado

[1]Categoria: docente, estudante, autodidata
[2]Se docente ou estudante

10 Noções básicas de alocação dinâmica de memória

10.1 O que é alocação dinâmica

Até agora, os programas utilizavam a memória do computador *estaticamente*: todas as posições de memória eram reservadas para as variáveis no início da execução do programa ou da função e, mesmo que não estivessem sendo mais utilizadas, continuavam reservadas para as mesmas variáveis até a conclusão da execução do programa ou da função. Um vetor global do tipo *float* com dez mil componentes, por exemplo, "ocupará" quarenta mil bytes de memória durante toda a execução do programa. Naturalmente, isto pode, em grandes programas, sobrecarregar ou, até mesmo, esgotar a memória disponível. No primeiro caso, há uma degradação na eficiência do programa; no segundo caso a execução do programa pode ser inviabilizada.

Os compiladores C permitem a *alocação dinâmica* da memória de tal modo que posições de memória sejam reservadas para variáveis no instante em que sejam necessárias e sejam liberadas (as posições de memória) para o sistema nos instantes em que não estejam sendo utilizadas.

A alocação dinâmica de memória pode ser feita através das funções *malloc()*, *calloc()* e *realloc()* cujos protótipos se encontram no arquivo *alloc.h* e são os seguintes

```
void *malloc(size_t Tam);
void *calloc(size_t NumItens, size_t Tam);
void *realloc(void *Bloco, size_t Tam);
```

Aí, *size_t* é um tipo de dado pré-definido, definido também no arquivo *alloc.h*, *Tam* é o número de bytes que se pretende alocar dinamicamente, *NumItens* é a quantidade de itens de *Tam* bytes que se pretende alocar e *Bloco* é um ponteiro que contém o endereço da variável cuja memória se pretende expandir em *Tam* bytes.

A função *malloc()* retorna um ponteiro para um bloco de *Tam* bytes até então disponível, a função *calloc()* retorna um ponteiro para um espaço de memória até então disponível capaz de armazenar *NumItens* objetos, cada um deles com *Tam* bytes e a função *realloc()* retorna um ponteiro para um bloco de memória com quantidade de bytes igual à soma algébrica da quantidade de bytes apontada por *Bloco* e *Tam*, podendo *Tam* ser negativo. Caso a quantidade de bytes pretendida não esteja disponível, as funções acima retornam *NULL*.

Como os ponteiros retornados são ambos do tipo *void*, eles devem ser moldados para poderem receber endereços de qualquer tipo de variável. Por exemplo, o programa

```
#include <stdio.h>
#include <alloc.h>
main()
{
int *v, t;
v = (int *)malloc(80);
if (v == NULL)
  printf("Memoria nao disponivel");
else
  {
  for (t = 0; t <= 40; t = t + 1)
            v[t] = t*t;
  for (t = 0; t <= 40; t = t + 1)
            printf("%d ", v[t]);
  free(v);
  }
}
```

armazena os quadrados dos quarenta primeiros números inteiros num vetor criado dinamicamente. A função *free()* libera para o sistema a quantidade de memória alocada para o seu argumento por uma das funções *malloc()*, *calloc()* ou *realloc()*.

Naturalmente, o leitor pode estar pensando que o programa acima não teria muita vantagem em

relação ao programa abaixo, onde o vetor v é criado estaticamente,

```
#include <stdio.h>
main()
{
int v[40], t;
for (t = 0; v <= 40; t = t + 1)
  v[t] = t * t;
for (t = 0; v <= 40; t = t + 1)
  printf("%d ", v[t])
}
```

De fato, os dois programas durante suas execuções utilizam oitenta bytes de memória. Haverá diferença se estes programas fizerem parte de um programa maior. Neste caso, o primeiro utiliza oitenta bytes apenas até a execução do comando *free()*, enquanto que o segundo utiliza os oitenta bytes durante toda execução do programa.

Naturalmente, também, o leitor pode estar se perguntando qual a vantagem do primeiro dos programas acima em relação ao programa

```
#include <stdio.h>
main()
{
int *v, t;
for (t = 0; v <= 40; t = t + 1)
  v[t] = t * t;
for (t = 0; v <= 40; t = t + 1)
  printf("%d ", v[t])
}
```

no qual o "tamanho" de *v* não foi fixado e portanto não há "desperdício" de memória. O que ocorre é que no primeiro se não houver memória disponível, o vetor *v* não é "criado", cabendo ao programador ajustar o programa para esta hipótese.

10.2 Armazenando dinamicamente um polinômio

A função *realloc()* aumenta a eficiência de utilização de memória pelo fato de que a quantidade de memória alocada para um vetor pode crescer à medida da necessidade. Para exemplificar, imagine a função abaixo que "lê um polinômio" p(x). Como se sabe, um polinômio $p(x) = a_0x^n + a_1x^{n-1} + ... + a_{n-1}x + a_n$ é identificado pelo seu grau e pelos valores de seus coeficientes. Se se pretende que o grau não seja dado de entrada, a questão é a mesma que ocorre quando se trabalha com uma relação de números: não se sabe a quantidade deles e, portanto, não se pode precisar o tamanho do vetor necessário. Uma solução é utilizar a função *malloc()* para inicializar um ponteiro e, à medida que os coeficientes são digitados, utilizar a função *realloc()* para expandir a memória necessária.

```
#include <stdio.h>
#include <alloc.h>
#include <string.h>
#include <stdlib.h>
int LePolinomio(int *p)
{
char *Coef;
int i;
printf("Digite os coeficientes ('fim' para encerrar)\n");
i = 1;
do
  {
  p = (int *)realloc(p, 2*i);
```

```
    gets(Coef);
    if (strcmp(Coef, "fim") != 0)
        {
        p[i - 1] = atof(Coef);
        i = i + 1;
        }
    }
while (strcmp(Coef, "fim") != 0);
free(p);
return (i - 1);
}
```

A "inicialização de p" deve ocorrer antes da chamada da função. Por exemplo, na função *main()* através do ponteiro declarado nesta função e que será passado para a função *LePolinomio()*.

```
main()
{
int *Poli;
int Grau;
Poli = (int *)malloc(2);
Grau = LePolinomio(Poli);
. . .
}
```

Observe que optamos por dar entrada nos coeficientes como *strings*. Esta opção foi feita para que pudéssemos usar "fim" como *flag*, já que a utilização de qualquer número com esta finalidade fica complicada já que qualquer número pode ser um coeficiente de um polinômio.

10.3 Listas

Para mais um exemplo de alocação dinâmica, apresentaremos um tipo de dado chamado *lista simplesmente encadeada*. Este tipo de dado pode ser definido como uma sequência de elementos ligados através de ponteiros com um número máximo de elementos não fixado *a priori*. Usualmente, cada elemento da lista é uma estrutura, com um campo contendo um ponteiro e os demais campos contendo os dados que o programa vai manipular. Associa-se um ponteiro ao primeiro elemento e o campo do tipo ponteiro do último elemento da lista tem valor NULL. O ponteiro que aponta para o primeiro elemento da lista indica o seu início e o valor do ponteiro da última estrutura da lista ser NULL indicará o seu final.

Para as funções que vamos apresentar (criação de uma lista de inteiros, exibição e remoção de um elemento desta lista) necessitamos definir a seguinte estrutura:

```
struct TElemento
  {
  int Valor;
  struct TElemento *p;
  };
```

Observe que na definição de uma estrutura é possível se definir um campo de tipo idêntico ao da estrutura. Este campo definirá um ponteiro que aponta para o elemento seguinte da lista e o campo *Valor* armazenará o inteiro.

Necessitamos também de duas variáveis globais:

```
struct TElemento *Inicio, *Prox;
```

a variável *Inicio* para apontar para o inicio da lista e *Prox* para apontar para o próximo elemento. É necessário observar que o ponteiro *Inicio* aponta para o último elemento que foi inserido na lista, enquanto que o campo *p do primeiro elemento da lista é igual a *NULL*.

Para criar a lista fazemos *Inicio* receber *NULL* e, dentro de uma estrutura de repetição, alocamos memória para o ponteiro *Prox*, damos entrada em *Prox.Valor* e fazemos o campo *p* de *Prox* receber *Inicio* e

Inicio receber *Prox* para que este ponteiro aponte sempre para o último elemento a dar entrada. Temos a seguinte função:

```
void CriaLista()
{
Inicio = NULL;
printf("Digite os numeros (-1 para encerrar)\n");
do
  {
  Prox = (struct TElemento *)malloc(5);
  scanf("%d",&(*Prox).Valor);
  if ((*Prox).Valor != -1)
          {
          (*Prox).p = Inicio;
          Inicio = Prox;
          }
  }
while ((*Prox).Valor != -1);
}
```

Para exibir a lista (ou realizar nela qualquer processamento), basta percorrê-la desde seu início (*Prox = Inicio*) até o seu final (*(*Prox).p = NULL*).

```
void ExibeLista()
{
Prox = Inicio;
while (Prox != NULL)
  {
  printf("%d ", (*Prox).Valor);
  Prox =  (*Prox).p;
  }
printf("\n");
}
```

Para deletar um elemento da lista é necessário que, quando ele for localizado, se armazene o ponteiro do elemento anterior a ele e o ponteiro que aponta para ele. Se o elemento a ser deletado for o primeiro, basta fazer *Inicio* apontar para o segundo elemento; se o elemento a ser excluído for outro elemento, basta fazer o ponteiro do elemento anterior para o elemento seguinte e devolver ao sistema a posição de memória ocupada pelo elemento a ser excluído.

```
void Deleta(int n)
{
struct TElemento *Ant;
Prox = Inicio;
Ant = Inicio;
while ((Prox != NULL) && ((*Prox).Valor != n))
  {
  Ant = Prox;
  Prox = (*Prox).p;
  }
if (Prox != NULL)
  {
  if ((Ant == Inicio) && (Prox == Inicio))
          Inicio = (*Prox).p;
  else
          (*Ant).p = (*Prox).p;
  free(Prox);
  }
else
  printf("Elemento nao esta lista \n");
```

}

10.4 Exercícios propostos

1. Escreva uma função que calcule a média de uma relação de números armazenada numa lista criada pela função *CriaLista()* acima.

2. Escreva uma função que insira um elemento numa lista ordenada de modo que a lista permaneça ordenada.

Observação

Propostas de soluções dos exercícios propostos podem ser solicitadas através de mensagem eletrônica para jaime@ccen.ufal.br com assunto RESPOSTAS LIVRO C, anexando o formulário abaixo devidamente preenchido.

Nome	Categoria[1]	Instituição[2]	Curso[2]	Cidade/Estado

[1]Categoria: docente, estudante, autodidata
[2]Se docente ou estudante

Bibliografia

Dijkstra, E. W., *A Discipline of Programiming*. Prentice-Hall. New Jersey, 1975.

Almeida, Eliana S. de et al.. ***AMBAP**: Um Ambiente de Apoio ao Aprendizado de Programação*. Anais do Congresso da Sociedade Brasileira de Computação, 2002, Florianópolis.

Evaristo, J. e Crespo, S, *Aprendendo a Programar Programando numa Linguagem Algorítmica Executável (ILA)*. Book Express, Rio de Janeiro, 2000.

Evaristo, J., *Aprendendo a Programar Programando em Linguagem C*. Book Express, Rio de Janeiro, 2001.

Evaristo, J., Perdigão, E., *Introdução `a Álgebra Abstrata*. Editora da Universidade Federal de Alagoas (EDUFAL). Alagoas, 2002.

Evaristo, J., *Programando com Pascal*. Segunda Edição, Book Express, Rio de Janeiro, 2004.

Knuth, D. E., *The Art of Computer Programming*, volume 2, *Seminumerical Algorithms* Addison-Wesley Publishing Company. USA, 1988.

Kowaltowski, T. & Lucchesi, C., *Conceitos Fundamentais e Teoria da Computação*. Anais do II WEI. Minas Gerais, 1994

Norton, P., *Introdução à Informática*. Makron Books. Sãp Paulo, 1996.

Rangel, J. L., *Os Programas de Graduação em Linguagens de Programação*. Anais do IIWEI. Minas Gerais, 1994.

Szwarcfiter, J. L. & Markenzon, *Estruturas de Dados e seus Algoritmos*. LTC Editora. Rio de Janeiro, 1994.

Wirth, N., *Algorithms & Data Structures*. Prentice-Hall. New-Jersey, 1986.

Índice remissivo